Welcome back. What was wrong with the first version of the clock? For one thing, trying to change the timing constants was very awkward, since the clock had to be stopped and the memory location holding the timing constant accessed with the DCM mode. This required that the clock be reset each time, a process that is time consuming to the point that we decided to add a SLOW/FAST feature to facilitate the tweaking of the program. Secondly, computers are meant to do things, to control their environment. With this in mind we decided to add the capability for turning other circuits on and off at times that we preselect. Since the computer talks to the outside world through the external I/O port that is brought to the edge connector, that means that drive circuits can be added to the system so that lights, motors, relays, sprinklers and other loads can be turned on and off at the times that you select. Such a system is called a programmable controller.

The process of developing the program for this system will provide an extensive exercise in using the stack memory for the storage of data. We will also get additional practice in the use of displays. The program is complex to the point that we need to find some way to save it without having to key it in manually each time it is to be used. This brings us to the subject of the tape recorder interface.

Mass Storage. By using a tape recorder with the Computer-in-a-Book we extend its capabilities tremendously. Now you have the ability to store your programs on cassette tapes. Between the programs of your own making and prerecorded tapes from Iasis, you can, in short order, assemble a tape library covering a wide variety of applications.

Since the subroutines that we develop for the programmable controller will be used throughout the rest of the book, this is a good time to introduce the subject of the tape recorder interface. That way you can save these subroutines on tape and use them in other programs without having to go through the agony of keying them in by hand every time the computer is turned on. The ia7301 is designed to interface with virtually any consumer audio cassette tape recorder of the inexpensive, portable variety. The only restriction is that the recorder have remote earphone and microphone jacks. The computer sends and receives audio signals through these jacks, and in this way communicates with the recorder. The tape interface program is contained in the system monitor and can be accessed directly through the use of the RT and WT keys.

Recording programs onto blank cassettes is quite straightforward since the recording level is independent of the volume control on the recorder. The computer delivers a sine wave of about 2Vpp to the microphone jack of the recorder. This is sufficient to saturate the tape and insures that the data can be read at a later time. When the tape is played back, the recorder sends an audio sine wave to the computer through the earphone jack. The amplitude of this signal is controlled by the volume control on the recorder, and since this control is an audio taper pot, a slight movement of the control produces drastic changes in the voltage output of the recorder. Although the ia7301 has been carefully constructed to ease the interface requirements, it will probably take a few moments of experimenting to find the optimum volume settings for any particular recorder and tape combination. This process will be described shortly.

Hooking up the Tape Recorder. Before trying the tape interface, it will be necessary to solder the tape recorder cables to the connector on the top of the computer. To do this, remove the connector from the computer PC board and warm up your soldering iron. The tape recorder uses the left end of the connector, at the opposite end from the power connections. The connector is a 56 contact connector configured as two rows of 28 contacts. Since the PC board is designed to make the top row of contacts electrically the same as the bottom row of contacts, from the standpoint of the connector it makes no difference whether you solder a wire to the top row or the bottom row. This means that you can think of the connector as a 28 contact connector with two solder lugs, one above the other, for each of the contacts.

Your computer has two cables provided for use with the tape recorder. Both cables have miniature phone plugs on one end that will plug into your recorder. The other end has two leads, each stripped and ready for soldering. One of these leads is the center portion of the cable; the other is made up of the shielded braiding twisted together to form a wire. This shielded braiding must be connected to the circuit ground as follows.

Consult Fig. 7-1 and solder the first cable to the connector. The center lead of the cable must be soldered to contact 28; the shielding must be soldered to the adjacent contact, 27. Again, it makes no difference whether you solder to the top row or the bottom row; in fact, the center can be soldered to one row and the shielding to the other. This cable will be used to plug into the earphone jack of the recorder. If your cables are identical, you should mark one of them to avoid confusion. This may be done either with a piece of tape or a knot at the tape recorder end of the cable.

Now solder the other cable to the connector. This cable, which will be used with the microphone jack, must be connected with the shielding to contact 26 and the center lead of the cable to contact 25. When the cables are soldered correctly, the shielded leads should be on adjacent contacts, 27 and 26. The center leads should be on 28 and 25. When the soldering operation has been completed, replace the connector on the system PC board. Remember, as you look at the computer with the connector at the top of the system, the power lines should be on the right end of the connector and the tape recorder cables at the left of the connector. Also be sure that as you slide the connector onto the mating fingers on the PC board, that the fingers line up with the mating contacts inside the connector. Failure to do this may damage the unit.

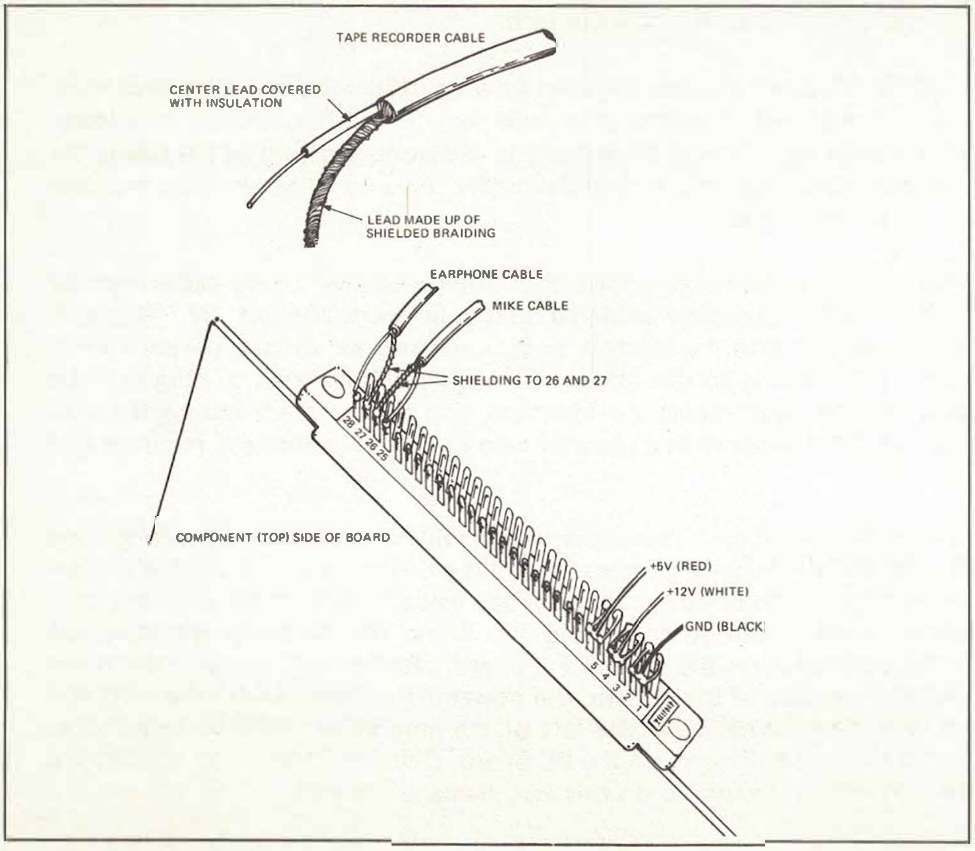


Fig. 7-1. Connecting the tape recorder cables.

Writing a Tape. We are now ready to write our first tape. Turn on the computer and plug in the tape recorder cables taking care that these are not reversed. The ia7301 is designed to operate with earphone jacks. These are not the same as AUX or auxilliary speaker jacks. If your recorder is equipped with an auxilliary speaker jack instead of an earphone jack, you may have to add a few inexpensive components to the edge connector to make the interface circuitry operate correctly. Assuming you have an earphone jack, read on.

Although the recorder may be operated from batteries, you will probably find that under heavy use the recorder will drain the batteries and cause unacceptable speed variations. For this reason, we recommend the use of an AC adapter whenever possible.

The ia7301 is designed to operate with most cassette tapes, but experience has shown that Maxell LN C60 tapes provide long life, trouble free performance and are relatively inexpensive ($1.83 each). Longer cassette lengths can cause speed variations which will cause the computer to read the tape incorrectly, so the shortest tape length you can purchase is the best. Since the shorter tapes are also less costly, there is an obvious incentive in purchasing the shorter lengths. We have tried other tapes such as the Ampex 350 which is available on a discounted basis in many hi-fi outlets, but have found that this tape eventually causes problems in trying to read programs. The frustration that accompanies the loss of a program due to tape failure is such, that you will doubtless find yourself, as we did, paying a few cents more for tapes and saving hours of hassle by not having to rekey programs.

Recording Formats. There are two basic formats for recording digital data onto tape. One is the file system which breaks data into blocks which can then be retrieved by the computer. This format is used when the data has an obvious type of "sort" attribute, such as names on a mailing list sorted by zip code. Each file could be made up of those names and addresses falling into a certain zip code category. That allows the computer to retrieve names from northern California, for instance, by examining the names in file 95 which contains all addresses in zip area 95000 to 96000. While this is a very desirable feature to have in a mass storage system, it requires that the recorder be able to count files while in the fast forward and fast reverse modes. The recorder must also be able to start, stop, reverse and high speed search under the direction of the computer. While the ia7301 is capable of being programmed to do this type of operation, the recorder must be solenoid actuated and that requires special digital recorder decks that run from $100 to $500 not including the interface electronics.

The requirement placed on the ia7301, that it must interface to inexpensive cassette recorders, obviously dictates some sort of compromise in data storage capability. The compromise means giving up the ability of searching out files of data that may appear randomly on the tape, and settling for the less ambitious mode of simply storing programs. This is still a very desirable feature of the unit. Simply stated, it allows us to store on tape any sequential portion of the memory that we wish. This can be data, programs, or both, but in any event, the entire contents of the tape are read in one continuous operation and loaded into the appropriate portion of the RAM memory. The recorder is then switched off and the program executed. If the program begins at 0100H, merely pressing CLR and EXC will start execution without having to initialize the program counter. How does the computer know where to lead the program as it reads it from the tape? Well, that question is best answered by considering the format that is used to record data onto the tape.

If you will allow us to beg the question of exactly how each binary bit is placed onto the tape, we promise to come back to it in a moment. Let's just assume that there are two audio tones, tone A for binary O and tone B for a binary 1. Since hex digits are really made up of O's and 1's, sending these two tones to the recorder in the proper sequence can cause the tape to be magnetized with the proper signals which can later be read back to the computer. While this is not the exact scheme used, it is close enough to allow us to pass on to loading programs.

The question of recording a tape presupposses that there is a program in the computer memory that is to be saved. That program obviously has a beginning and an ending. Most of our programs start at 0100H and work up to some upper address that represents the last instruction or data location. Some programs may use locations under 0100H for data storage or subroutines. In any event there is always a lower address that is the lowest memory location ever used by the computer and an upper location beyond which the computer never goes. The low address is defined as the starting address and the high address as the ending address. Understand that we are not now speaking of the first and last instructions executed, but rather the lowest and highest memory locations used by the computer to store the program. As the first part of the tape write operation, it is necessary to specify to the computer exactly what part of memory is to be recorded onto tape. We do that by using the DCR key to load the starting and ending addresses into the D,E,L, and H registers. The H/L register pair is always used for the starting address, with the most significant digits going into the H register. Likewise, the D/E pair is always used for the ending address with the most significant digits going into the D registers. This is consistent with the programming practice followed to date, that is, the D and H registers are the high order registers; the E and L registers are the low order registers. To cause the memory from 0100H to 03FFH to be stored on tape, load D with 03H, E with FFH, H with 01H and L with 00H. Of course, we can ignore the starting and stopping points of our program by always loading the entire memory, from 0000H to 03FFH, but this requires about three minutes and if we can shorten the recording and playback time by only storing portions of the memory, less time is wasted.

Once the computer's registers have been instructed to place a portion of memory on to tape, you need only to start the recorder tape into motion and press WT , Write Tape. Obviously if we wish to record tape, the recorder must be in the Record mode. However this is done, it is up to the user to set the tape into motion and press the WT key. The monitor program in the computer then takes over and the rest of the operation is entirely automatic.

But what if you're a mite clumsy in hitting the WT key after starting the recorder? Will the delay cause a problem? No, because the recording process does not really begin until WT is pressed, and even if we do the operation in reverse, that is, press WT and then start the recorder, you have about 25 seconds of leeway. When WT is pressed, the computer responds by generating an audio tone of about 600Hz and sending it to the recorder microphone jack, In fact, this is tone A and is the lower of two tones. This tone lasts for 25 to 30 seconds and represents a tape leader to ease the problem of the user having to operate both recorder and computer controls at once. This is where the leeway just mentioned is derived, data does not start being send to the recorder until the leader tone is complete.

When the 30 second leader is finished, the computer begins sending data to the recorder which is loaded onto the tape. The first data is not, as you might expect, the first of the program insturctions at the starting address. Rather, it is the starting address itself. This is immediatley followed by the ending address. The computer is able to do this, because we had earlier stored these two addresses in the D/E and H/L register pairs. Now it has only to send them to the recorder. This being done we have given the system the ability to later read the tape and know where the data it finds there is to be stored in memory without having to bother keying in the addresses as we did before the Write Tape operation. Once the two address have been loaded onto the tape, they are followed by the data itself, beginning with that at the starting address.

As each two digit data word is sent to the recorder, the computer does a strange thing. It adds the data to the data already sent, so that a running total of all data is kept as the write operation procedes. The second data word is added to the first, the third is then added to this sum to form a new total, the fourth is added to the first three, and so on. This running total is termed the *checksum* and it is used to check for errors during playback.

At the same time the computer is sending data to the recorder, and doing the additions to update the running total that is the checksum, it performs a third operation. It counts the number of data words that it has sent to the recorder. This count is added to the starting address and the result is compared to the ending address. When a match occurs, the computer knows it has completed sending the specified data to the recorder. There is only one final operation; the checksum is sent to the recorder as well. For the sake of simplicity, the computer ignores any carry out from checksum additions. This means that the checksum is only two hex digits in length. Of course, the number represented by those digits bears no relationship to the actual total that would have resulted had the carries been taken into consideration. An analogous situation would be adding up your purchases at the supermarket to check the addition of the check out clerk. The hard way would be to add all of the digits, so that if your total of $76.49 agrees with the clerk, either you both made the same mistake or, what is more likely, you are both correct. A simpler way, similar to the checksum, is to simply keep track of the cents and omit the dollars. If your "penny count" is .49 and the clerk claims the total is $76.48, one of you is obviously wrong.

Ah, but you argue that simply dropping a digit in the dollar column would not produce a detectable error this way. That is, if during your shopping session, you inadvertantly added an item costing $1.23 as .23, omitting the dollar, your penny count would match the total generated by the clerk, but there would still be a discrepancy. This is true, but in the case of the computer checksum, the data is never more than two digits in length. It would be as though we had never purchased a item costing more than 99 cents. This is a considerably different situation than simply dropping a higher order digit. Now, an error could only go undetected if a second, offsetting error was made. With 256 possible combinations of the two hex digits, our error checking scheme should detect an error successfully over 99.5% of the time.

During the entire time that the computer was sending data or leader to the recorder, the LED in the lower left of the computer labeled WRITE TAPE was lighted. The instant the recording operation is completed, this LED will be turned off and the displays will come up in the DCM mode. You may now shut off the recorder; your program is on the tape.

In order to experiment with the volume settings on the recorder we need a tape that contains known data. It is then relatively straight forward to check the contents of the memory after reading the tape and counting the errors. In this way we can optimize the settings of the volume control to determine the min-max positions that will cause error free operation. To get a tape with known data requires that the corresponding portion of memory contain known data. Since for statistical purposes we need a large block of memory in which to read and later reload data, we will generate a short program that saves us the time of having to key in data into the hundreds of memory locations used in this test. The program is stored in low order RAM locations starting at 0020H. It operates on the much larger block of addresses between 0100H and 03FFH, 768 locations in all. We execute these programs by loading the program counter with 0020H using DCR and then pressing EXC . There are actually two programs; the one at 0020H loads the memory locations with 55H and the one at 0040H checks them after reading the tape and counts up errors. This will become clearer as you examine the two programs listed below.

TO WRITE DATA "55H"

```
0020    21    LXI H,0100H           ;Point H/L register pair to the
0021    00                          ;starting address 0100H.
0022    01                          ;
0023    36    MVI M,55H             ;Send data word 55H to the memory
0024    55                          ;Location pointed to by H/L.
0025    23    INX H                 ;Increment the pointer.
0026    7C    MOV A,H               ;Bring the most significant two
0027    FE    CPI      04H          ;address digits to the accumulator
0028    04                          ;and compare to 04H.  A match means
0029    C2    JNZ      0023H        ;we have reached 0400H and the
002A    23                          ;program is complete.  Otherwise
002B    00                          ;loop back to 0023H and repeat.
002C    FF    RST 7                 ;Program stopper..
```

COUNT ERRORS

```
0040    21    LXI H,0100H        ;Point H/L register pair to the
0041    00                       ;starting address 0100H.
0042    01                       ;
0043    01    LXI B,0000H        ;Set up B/C pair as error counter by
0044    00                       ;loading both with zeroes.
0045    00                       ;
0046    7E    MOV A,M            ;Fetch data from memory location
0047    FE    CPI     55H        ;pointed to by H/L.  Compare to
0048    55                       ;55H.  If error, go to 0060H.  Otherwise
0049    C2    JNZ     0060H      ;increment H/L pair to point to next
004A    60                       ;location.  Check to see if we have
004B    00                       ;reached 0400H.  If so, the program
004C    23    INX H              ;is complete.  Otherwise, loop back
004D    7C    MOV A, H           ;to 0046H and repeat.
004E    FE    CPI     04H        ;
004F    04                       ;
0050    C2    JNZ     0046H      ;
0051    46                       ;
0052    00                       ;
0053    FF    RST 7              ;Program stopper.

0060    0C    INR C              ;An error has been found.  Increment
0061    C2    JNZ     004CH      ;the C register.  If this results
0062    4C                       ;in a zero in C, it must be because
0063    00                       ;C contained FFH and we are now seeing
0064    04    INR B              ;a carry.  If so, increment B to
0065    C3    JMP     004CH      ;record the carry and return to
0066    4C                       ;the main program loop at 004CH.
0067    00                       ;
```

Load these programs into the computer. All of the instructions are known to you except the MOV A,H which loads the accumulator with the contents of the H register, and INR B and INR C which cause the B and C registers to be incremented respectively. The comments that accompany the two programs should be clear. The first is executed as described earlier; the program counter is loaded with 0020H by using DCR , and then executed by pressing EXC . Don't press CLR first, it'll reload the program counter with 0100H and execution will begin at that point in error. The program points the H/L register pair to 0100H, sends the data word 55H there, increments the pointer and sets up a loop so that the entire memory from 0100H to 03FFH is loaded with 55H in every location. On each pass through the loop, tlhe contents of H register are moved to the accumulator and check to see if H has reached 04H. This happens as the pointer is incremented from 03FFH to 0400H which is the end of the program. When that happens, the program falls out of the loop and reaches the FFH program stopper. You can check the results by using the DCM mode to manually check some of the memory locations to be sure they all contain 55H.

The second program is used to scan the same memory segment to be sure each location contains 55H after the first program is executed. It too points H/L at the starting location 0100H. It also sets up the B/C register pair as an error counter by initializing both registers to zero. Each memory location is brought to the accumulator and checked to be sure it contains 55H. Each time this happens the H/L pair are incremented to point to the next location and checked to see if the ending address of 03FFH has been reached. If so, the program is terminated with a FFH program stopper. If not, the program loops back and checks the next location. This looping action containing 55H, a jump to 0060 occurs. This portion of the program increments the B/C register acting as the error counter. Each time the C register reaches 00H, a carry out is added to the B register with an INR B instruction. If you execute this program by loading 0040H into the program counter with the DCR key and the press the EXC key, it will count the errors from the first program. Since the program is terminated with an FFH, it will end by going into the STEP mode. In that mode, you can examine the B/C register pair using DCR and see how many errors are in the memory. If you loaded and executed the programs correctly, there should be none.

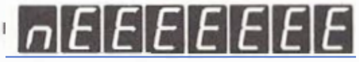
Now, let's write our first tape. Without modifying any of the memory locations loaded with 55H by the first program, set the H/L pair to 0100H and the D/E pair to 03FFH with the DCR key. About half of the problems with the tape interface occur right here; failure to correctly load the registers with the starting and ending addresses. Remember, H is loaded with 01H, L with 00H, D with 03H and E with FFH. When this has been completed check to be sure the tape in the recorder is completely rewound. Then set the tape in motion in the RECORD mode. Immediately press WT on the computer to start the data transfer. You should see the WRITE TAPE LED light and the displays go blank. If you have a watch, you can time the operation . After slightly over two minutes have elapsed, the LED will go out and the displays will come back up in the DCM mode. You have written your first tape.

Reading Tape. The tape you have just written will become the test tape for the calibration process that follows. Since the tape contains known data, we can read it back into the computer and then use the program at 0040H to count the errors. Errors result from an improper setting of the volume control on the recorder. By repeated playings of the test tape, we can find that range of volume control settings that allow error free operation of the system. The range will vary from recorder to recorder. We use an inexpensive ($40) unit from Radio Shack labeled CTR-29, and on that recorder error free operation occurs over about 75% of the volume control range. As you can see the system is relatively tolerant to volume setting and once calibrated can be expected to read and write tapes with no problem at all. In fact the system is designed to operate with signal ouputs from the recorder anywhere between .6vPP and about 7.0vPP. In this respect, the cheaper recorders are actually easier to work with than the more expensive units. That's because the better recorders have more output and a larger portion of the volume control setting will result in ouputs in excess of 7.0vPP.

For a first pass at reading tape, set the volume at maximum and set the tape in motion in the PLAYBACK mode. Immediately after setting the tape in motion press the RT key, Read Tape. The rest of the operation is automatic.

When the tape is first set into motion the ouput from the recorder may be slightly spurious to say the least. To avoid the possibility of the computer misinterpreting this startup noise as data, the monitor program directing the computer sits and counts leader pulses for about 15 seconds. This gives the recorder lots of time to settle into speed and let any noise die out. It also gives you a chance to get your finger from the PLAY button on the recorder to the RT key on the computer. After the first 15 seconds the computer begins to look for data. The format used for the recording of each bit insures that the first bit always begins with tone B, it is immediately obvious to the computer where the leader ends and data begins. Of course, data here refers to the starting address in memory that is destined to receive the data coming off the tape, since that was the first thing that was recorded when the tape was written. This is followed immediately by the ending address which defines the upper limit of the recipient memory segment. As the data that follows this second address comes in from the recorder, the computer loads it into memory. It also keeps a running count of the locations loaded so that it knows when the memory block is full. Finally the computer generates a checksum in the same fashion it did during the tape writing process. You and the check out clerk are about to compare notes.

That's right. When the last location in the specified memory block is loaded from the tape, the computer is aware of the fact since it has been counting locations during the entire process. And if the last memory data just came through the cable, then what must follow it is obviously the checksum that the computer placed on the tape as its last act of Tape Write. As that checksum arrives the computer compares it to the checksum it just generated during the read operation. If the two match, all is well and the program is terminated by turning off the READ TAPE LED and bringing the displays up in the DCM mode. If, however the checksum comparison failed, the displays come up as shown below:



The tape system therefore has the ability to check itself. Of course, it can't locate which address failed to receive the correct data, but it can flag the fact that at least one location is incorrect. And as it turns out, that is quite sufficient.

If you followed our earlier instructions to start the recorder, it has been reading data into the computer as you have been reading this page. Since it takes exactly the same amount of time to read a tape as it did to record it, in about two minutes the read operation should be completed. One of three things will now happen. The tape read will be completed with no errors, the checksum comparison will be successful, and the displays will come up in the DCM mode. The second possibility is that the checksum comparison will fail and the displays will come up with E's signifying an error has been detected. The third case, which is the most disturbing is that the signal from the recorder will be so far from the correct level that the computer never even read the leader correctly. You can tell when this happens because when the two minutes is up the computer will remain in the Read Tape mode with displays blanked and the one LED lighted.

Let's assume the worst. The computer is playing dumb and not responding to the recorder at all. Rewind the tape, reduce the volume control setting by about 1/3 and repeat the process. By a trial and error you should be able to find a point on the volume control setting where the computer comes up either in the DCM mode or with E's in the displays at the end of the two minute period. When this occurs, you're very close to having the system operative. Continue to adjust the volume control until you have located the maximum setting at which the computer will read the tape and come up in the DCM mode. Since the tape that you originally wrote for calibration purposed only stores the locations from 0100H to 03FFH, the programs that you loaded into 0020H through 0067H will be unaffected by the writing and reading of tape. Assuming you have not turned the computer off, they should still be in memory. When the computer exits from the tape read program into either DCM or the error display, you can use the program at 0040H to check the accuracy of the tape read operation. Use DCR to load the program counter with 0040H and then execute that program with the EXC key. When the program is completed, it will be terminated in the single step mode and you can use DCR to examine the B and C registers which contain the number of errors. Assuming the diplays came up in DCM, the B/C register pair should contain zero errors. If, however, the displays come up with E's the B and C registers will contain the number of errors. You can use this program to check the volume setting and find the maximum setting that gives reliable read operation. As you reduce the setting you should see the number of errors go down until it reaches zero and the displays are coming up in the DCM mode. Note this point on the control. In the case of our Radio Shack units, the control does not have the common 1 to 10 markings so we use a small piece of masking tape. This is the maximum end of the usable range.

We will now find the minimum usable setting. Using the same process just described, continue to reduce the volume setting and reading the tape, At some point the unit will stop coming up in DCM and go back to the E type display. This is the minimum setting and you should mark it. Pick a point in between and you should have error free operation 100% of the time. On our unit the range is from about 4 to 9½ (estimated since there are no markings on the dial).

Although the ia7301 is advertised as working with recorders having an earphone jack you can usually make it work with speaker jacks by adding a few components right to the edge connector. The circuit for doing this is shown in Fig. 7-2. Before you make the modification, try the system as you received it. Computers produced after the early models had an internal modification made at the factory to make them work with speaker jacks as well, although no guarantee is made that it will work with all recorders using speaker jacks.

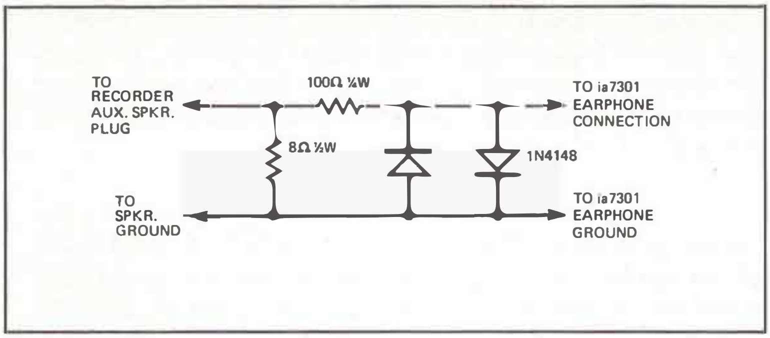

TO RECORDER AUX. SPKR. PLUG
100Ω ¼W
TO ia7301 EARPHONE CONNECTION
8Ω ¼W
1N4148
TO SPKR. GROUND
TO ia7301 EARPHONE GROUND

Fig. 7-2 Adding these four components to the ia7301 edge connector will usually make it operable with recorders having auxilliary speaker jacks instead of earphone jacks.

To review the tape operation, a program is stored on tape using the DCR mode to load the H/L registers pair with the starting address of the block of memory to be stored on tape, the D/E pair with the ending address of the memory block, the tape set into motion in the RECORD mode and WT , Write Tape, pressed. When the displays come back up in DCM , the operation is completed. To read the tape back at a later date, set the tape in motion in the PLAY mode and press RT , Read Tape. For both of these operations the volume control should be set at the optimum setting as determined by the calibration process just discussed.

The actual recording format used by the ia7301 uses a combination of two audio tones generated by the ia7301. The low frequency is about 1200Hz, the high is exactly twice the lower, or 2400Hz. Each bit of data recorded is split into three equal times. During the first third, a high frequency is recorded regardless of whether the bit is a 0 or a 1. During the last third, a low frequency is recorded regardless of whether the bit is a 0 or a 1. During the middle or second third the bit information is recorded onto the tape, low frequency for a 0 and high frequency for a 1. This means that every bit begins with a high frequency and ends with a low frequency regardless of whether the bit is a 0 or a 1. The process begins by writing about 30 seconds of low frequency onto the tape followed by the first data which is the starting address of the memory block being recorded. Since each bit always begins with a high frequency, the computer will be able to distinguish between the low frequencies of the leader and the beginning of the first bit of data. This two-tone split bit time gives a high degree of speed tolerance to the recorder and insures that tape imperfections will not cause program errors.

Practical Programming. Now that we have the ability to save and reload our programs from the recorder, we can write much more elaborate programs. The first of these will be a much more sophisticated version of the digital clock we worked with in the last chapter. Before we do that, however, a few words are in order about the programming methods used in actual practice.

The previous chapters have stressed the need for reducing each program to the minimum amount of program storage possible. This was done because memory costs money and shortening a program reduces the amount memory needed. It is possible to go too far with this method. Recent advances in the art of making semiconductor memories have reduced their cost to the point where the programmer's time may actually be more expensive than the memory used to store his program. An age-old industry rule of thumb maintains that a fully debugged and well documented program costs on the average of $10 a line to generate. At that rate, it costs $160,000 to fill a 16K RAM board with program!

Keep in mind that the $10/line rule is for commercially saleable programs that are totally debugged and completely documented. A program done for your own use with minimum amount of documentation would not cost anywhere near that number no matter how you value your time. Likewise, in the situation where you are given a fixed amount of memory, as in the case of the ia7301, it is foolish to spend any amount of time trying to shorten the program beyond the necessity of fitting into the allocated memory block. However, for those applications where the program is going to be loaded into PROMs and sold as part of a microcomputer product, the rule is a valuable one. If 10 systems are going to be sold with a resident program occupying 16K of memory, each of those systems must have $16,000 of programmer's time allotted to it. If 1000 systems will be sold, that number is reduced to $160/system. Obviously the amount of time required to shorten a program goes up as more and more debug time is spent, since the earlier ineffeciencies will be found rather quickly, but the more subtle ones will take more and more time. It is very important that the programmer have some idea of how many systems will be sold so that he can accurately guage when to stop beating a program to death. This is usually stated at the beginning by a statement like "Don't spend more than 30 minutes per line of program eliminated." The programmer then knows that he should not pursue a change that will shorten the program by 25 lines if more than 12 hours of his time are required.

It is important to realize that practical programming never procedes by the programmer going to a piece of paper and setting down the entire program, start to finish. It is done by writing short program segments, testing them, adding others, testing the combination, and so on. Then after the entire program is working, the process of shortening it begins. To give you a feeling for the entire process we'll use the programmable controller as an example of how a good programmer attacks the problem.

Delay. As part of the programming exercise we will also develop a set of utility routines which we can use in all of our programs. These can be viewed as an extension of the system monitor. The utility routines will be stored in addresses 0300H to 03FFH in the form of callable subroutines that can be used by any of our programs.

The first of these utility routines will be the DELAY subroutine that we have already used in several of our programs. Since we will need it for the programmable controller program, let's write and debug it first. So far we have treated this subroutine as a program loop whose length is determined by a delay constant built right into the program. That's fine for the application using that particular delay, but it means that if a program required two delays of differing lengths, we would have to write two entire DELAY subroutines, each with a different delay constant. Of course, we could have the calling program go in and modify the DELAY subroutine by changing the delay constant before it performed the CALL, but that would make it very easy to create errors. Let's rewrite DELAY so that it can be used with any delay constant just by the way we CALL it in the main program. For talking puposes, we can think of the process defined by CALL DELAY, 1500H, as causing the DELAY subroutine to be executed 1500H times. This would certainly be convenient for the programmer since he could use DELAY as often as he wanted with as many different delay constants as he wanted.

If this all sounds great, we should point out the problems with this type of program. Writing the hex digits for a CALL instruction requires three lines, one for the CALL itself and two for the address of the subroutine. That subroutine must be terminated with a RET instruction that returns program execution to the main program at a point immediately after the three line CALL. In our case that line is going to be occupied by the delay constant, so that the RET will cause the program control to return the delay constant and try to execute it as an instruction.

Suppose that DELAY were located at 0300H and we tried to CALL it in the manner just described with a delay constant of CD00H.

```
0100    CD    CALL DELAY (0300H)
0101    00
0102    03
0103    CD    RELAY CONSTANT (CD00H)
0104    00
0105    01    LXI B, XXXXH
```

In this program, DELAY will be CALLed but when the RET that terminates that subroutine is executed, the return will be to location 0103H which is not really an instruction but the delay constant for the subroutine. Since the computer does not know this, it will try to execute what's at 103H as an instruction. Since CDH happens to be the hex code for RET, this is what the computer will execute which will foul things up completely. What we really want to happen is for the RET to cause program execution to return to 0105H which is actually the next instruction of the program.

Parameter Passing. The subject that we have been discussing is that of passing parameters to a subroutine, and it is a very important subject to programming. Subroutines frequently require one or more parameters. Sometimes these are in the form of data requiring only two hex digits, but they are frequently in the form of addresses that require four hex digits. Occasionally several of each type are required. Generally speaking, there are three ways of passing parameters to subroutines.

The first method is to load one or more of the working registers with the parameter and then CALL the subroutine. This method is usually the most efficient in terms of program length since the subroutine is entered with the required data already in the working registers. It is usually not suitable for complex subroutines that require the full use of the working registers since at least some of these are tied up with the parameters being passed. This method has the further drawback that it limits the use of the subroutine, because to use it the registers to be used for parameter passing may first have to be PUSHed onto the stack, then loaded with the desired parameter, and later restored to their former values.

The second method is to reserve certain memory locations for parameter passing. This is much more flexible since it allows a subroutine to be entered with the full use of all of the working registers. It has the single disadvantage of forcing the programmer to keep very accurate track of exactly which memory locations are reserved for which parameters. For a set of utility routines it is probably best to restrict the system as little as possible. For the main program this second method is excellent; the restriction that the programmer need to keep accurate track of the use of memory locations is trite since the first sign of a good programmer is accurate and copious documentation. If the second method is to be used, keep in mind that the memory location containing the parameter to be passed will be accessed each and every time the subroutine is CALLed. Great care should be used so that the location is loaded prior to each subroutine CALL. Failure to do this may cause errors since the various loops and nesting of subroutines within the program may cause the subrouinte to be CALLed at unexpected times.

The third method is to pass the parameter to the subroutine right in the program by following the CALL with the data as in the discussion of DELAY. For utility routines this is doubtless the best method since it puts the fewest restrictions on the programmer, ties up no working registers or memory locations and requires no documentation. It is also the trickiest.

DELAY. Our goal, then, is to write the DELAY subroutine so that it treats the contents of the two memory locations following the CALL address as the delay constant for the subroutine. That means that we have to find some way to fetch that data to the registers within the subroutine. Pretend for a moment that the little program segment we presented a couple of pages ago to illustrate the difficulties of DELAY is, in fact, the actual program that is to CALL DELAY. The value of the delay constant to be passed is in locations 0103H and 0104H. Since we are going to use this to test our subroutine, we need a way of stopping our program, so insert a RST 7 at 0105H. Load this program into the computer.

```
0100        CD        CALL DELAY (0300H)
0101        00
0102        03
0103        00        DELAY CONSTANT (004F)
0104        4F
0105        FF        RST 7
```

OK, that's the easy part. Now let's write the actual DELAY subroutine that will be stored at location 0300H. Our first task is to fetch the delay constant that is presently stored in locations 0103H and 0104H. This is not the simple matter of using an LDA instruction, for instance, since DELAY may be CALLed many times from many places. Each time the delay constant will follow the CALL, and since that CALL may be anywhere in memory, it follows that we really have no way of knowing in advance where the delay constant will be. Fortunately the computer knows!

At the risk of boring you with another pass over the role of the stack pointer, we need to examine that register during the subroutine CALL since it holds the key to fetching the delay constant. When a CALL is executed, the return address must be stored in memory so that at the completion of the subroutine, program control can be returned to the CALLing program. Just storing the return address is not enough, since we have to keep track of where in memory that address is stored. This is the job of the stack pointer; it contains the address of of the memory location that holds the return address. In addition to saving return addresses, the stack memory can be used to save the contents of registers that must be used in the course of the subroutine. This is done through the use of the various PUSH and POP instructions which load and unload registers into memory locations. Each time this happens, the stack pointer is incremented or decremented to record exactly where the stack memory begins and ends. As more data is saved in the stack it must grow and the stack pointer is moved accordingly. Later as the data is retrieved those memory locations become free for storing other addresses and data. The stack memory shrinks and the stack pointer records the fact. Since the stack grows downward, that is, data is added to the bottom of the stack rather than the top, adding data with PUSH instructions causes the stack pointer to be decremented.

As we execute the CALL as locations 0100H, 0101H and 0102H to implement the delay, the return address will be saved on the stack at whatever locations the stack pointer tells us are open. In order to follow what is about to happen, load an E3H into location 0300H, a 22H into the H register and a 11H into the L register using the DCM and DCR modes. Clear the computer and press STEP .

Enter:	See Displayed:
CLR STEP	⊢0300-E3
DCR NXT NXT NXT NXT NXT NXT	uL----11
NXT	uH----22

There are the 11H and 22H that we stored in L and H earlier.

NXT	uSP-00FE

As we might have expected, the stack pointer now contains 00FEH reflecting the effect of the routine CALL instruction. When that CALL was executed, the stack pointer was decremented by one from its normal 0100H to 00FFH and the most significant two hex digits of the return address stored there. The stack pointer was then decremented once more to 00FEH and the two least significant hex digits of the return address stored there. We can prove this to ourselves by using DCM to examine those locations.

Enter:	See Displayed:
DCM 0 0 F E NXT	n00FE-03
NXT	n00FF-01

Since the computer stores the most significant digits in 00FEH, these locations apparently now contain the address 0103H. This is consistent with what we would expect since 0103H is the return address, that is the address of the nest location following the three line CALL instruction at 0100H, 0101H and 0102H. Now execute the instruction at 0300H by pressing STEP once more.

Enter:	See Displayed:
	n00FF-01
STEP	t0.301-

Now examine the registers and memory locations again.

DCR NXT NXT NXT NXT NXT NXT	uL----03
NXT	uH----01
NXT	uSP-00FE

The hex code at 0300H is that of the XTHL instruction, Exchange Stack. When this instruction is executed, the contents of the L register are exchanged with the contents of the memory location whose address is held in the stack pointer. The contents of the H register are exchanged with the contents of the memory location whose address is one greater than that held in the stack pointer. In the example in our computer right now, we saw that executing the XTHL instruction at 0300H caused the H/L register pair to contain the return address that was formerly pointed to by the stack pointer. The contents of the stack pointer itself does not change. Before the XTHL instruction the H register contained 22H and the L register contained 11H; they now contain 01H and 03H, respectively. Use DCM to examine the memory locations 00FEH and 00FFH.

Enter:	**See Displayed:**
DCM 0 0 F E NXT	n00FE-11
NXT	n00FF-22

From the standpoint of the H and L registers, the XTHL instruction is very similar to the PUSH H instruction we have already discussed. Both save the prior contents of the H and L registers on the stack. In the case of the PUSH H instruction, however, the registers are then free to be used by the program. In the case of the XTHL, the H and L register are loaded with the address that is on the top of the stack. In this case, what is on top of the stack is the return address that was loaded there when the subroutine CALL was executed. Normally, this is the address that program execution should return to when the subroutine is completed. In the case of the CALL DELAY segment at 0100H through 0104H, that address is that of the first location containing the delay constant.

Many of the instructions we have studied so far have had counterparts for the other registers as well. This is not true of the XTHL instruction which only operates with the H/L registers. That and the ability to directly transfer data from register to memory by a MOV X, M make the H and L registers the strongest and most useful of the register pairs in the CPU. These are followed by the D/E pair. The B/C registers are the weakest of the set.

Our delay subroutine will also use the B, C registers as well as the accumulator and the flags, so it will be necessary to save them on the stack with PUSH instructions. Since we will want to monitor the action of the registers as they are saved in the stack, it is a good idea to load those registers with data that is easily recognizable. Let's start over, first loading the registers. Our favorite method is to store AA in the accumulator, FF in the flag register, BB in the B register, etc. This works until we get to the H and L registers, and as we have already seen, we can load 11H into the L register and 22H into the H register. By consistantly using these values to check the ability of our programs to store and retrieve data in the registers will save you much time and confusion.

Enter:	See Displayed:
DCR A A	uA----AA
NXT F F	uF----FF
NXT B B	ub----bb
NXT C C	uC----CC
NXT D D	ud----dd
NXT E E	uE----EE

NXT L L	uL----11
NXT H H	uH----22
NXT	

We also want to load the rest of the DELAY subroutine and then we'll step our way through it, discussing the various data transfers as they take place.

```
                                   DELAY

0300     E3     XTHL               ;Load H/L with address or delay constant.
0301     C5     PUSH B             ;Save B/C and PSW in stack.
0302     F5     PUSH PSW           ;
0303     46     MOV B, M           ;
0304     23     INX H              ;
0305     4E     MOV C, M           ;
0306     0B     DCX B              ;delay constant
0307     79     MOV A, C           ;Move least significant digits of
0308     B0     ORA B              ;delay constant to accum and check
0309     C2     JNZ 0306H          ;for zero contents.  If not zero,
030A     06                        ;loop back.  If zero, restore registers
030B     03                        ;and return.
030C     F1     POP PSW            ;
030D     C1     POP B              ;
030E     23     INX H              ;Correct return address.
030F     E3     XTHL               ;
0310     C9     RET
```

Clear the computer and press STEP four times to execute the XTHL and two PUSH instructions.

Enter:	See Displayed:
CLR STEP STEP STEP STEP	⊢0303-46
DCR	uA----AA
NXT	uF----FF *
NXT	ub----bb
NXT	uC----CC
NXT	ud----dd
NXT	uE----EE
NXT	uL----03
NXT	uH----01
NXT	uSP-00FA

Obviously, the two PUSH instructions do not affect the registers themselves since they apparently still contain their original contents. They have, however, been duplicated in the stack memory so the registers are actually free to be used for other things. Use DCM to examine the stack memory starting at 00FAH and working up.

Enter:	See Displayed:
DCM 0 0 F A NXT	n00FA-FF *

That location now contains the duplicated contents of the flag register. PSW was the last PUSH instruction executed so the next location should contain the contents of the accumulator.

NXT	n00Fb-AA

Prior to the PUSH PSW we saved the B and C registers on the stack.

NXT	n00FC-CC
NXT	n00Fd-bb

* IDIOSYNCRACY: CPU'S FROM SEVERAL SEMICONDUCTOR MANUFACTURERS ARE USED IN THE PRODUCTION OF THE ia7301 COMPUTER-IN-A-BOOK. THE VARIOUS DEVICES DIFFER IN ONLY ONE RESPECT – THE WAY THEY TREAT THE VARIOUS FLAGS. THE INFORMATION ON PROGRAMMING PRESENTED IN THIS COURSE IS DESIGNED TO BE INDEPENDENT OF WHICH CPU IS USED. ALL OF THE PROGRAMS WILL WORK NO MATTER WHICH CPU IS IN YOUR SYSTEM. THERE IS A SLIGHT DIFFICULTY, HOWEVER, IN THE VISUAL PRESENTATION OF THE FLAG REGISTER. LOADING FFH INTO THE FLAG REGISTER VIA THE DCR MODE AND THEN STEPPING AN INSTRUCTION WILL CAUSE A FFH TO BE DISPLAYED WITH ONE CPU AND A D7H WILL BE DISPLAYED IF THE SYSTEM USES THE OTHER CPU. THIS IS BECAUSE THE FLAG REGISTER DISPLAYS THE CPU FLAGS; SINCE THERE ARE ONLY FIVE FLAGS, BUT THE REGISTER HAS SPACE FOR EIGHT, THE CPU STUFFS THE THREE UNUSED

LOCATIONS WITH NULLS. FOR ONE PROCESSOR THESE NULLS ARE STUFFED WITH LOGIC ONES, ANOTHER CPU WILL STUFF THEM WITH A MIXTURE OF ONE'S AND ZEROES. THUS THE REGISTER, AS DISPLAYED, APPEARS DIFFERENT DEPENDING UPON THE STUFFING SCHEME OF THE CPU USED. THIS WILL BE EXPLAINED MORE FULLY IN CHAPTER NINE. IF THE STEPS ABOVE CAUSE A D7H TO BE DISPLAYED INSTEAD OF THE FFH THE TEXT SHOWS, PLEASE BEAR WITH US UNTIL CHAPTER NINE.

Finally, locations 00FEH and 00FFH should contain 11H and 22H which were the original contents of the L and H registers.

NXT	n00FE-11
NXT	n00FF-22

At this point in our program we have succeeded in saving the pertinent registers in the stack; the D and E registers were not saved since they will not be used in the subroutine. In addition, the H/L register pair now contains the address of the most significant two hex digits of the delay constant. The instruction at 0303H, MOV B, M, causes the memory location pointed to by H/L to be loaded into the B register. INX H increments this address so that the pair now points to the next location which contains the least two significant hex digits of the delay constant. MOV C, M loads the C register with this data so that the B/C register pair now contains the complete delay constant. We can check this action by using the STEP key and DCR to examine the various registers.

Enter:	See Displayed:
STEP STEP STEP	⊢0306-0b

DCR NXT NXT

NXT

```
ub----00
uC----4F
```

With the delay constant securely in the B/C register pair we are ready to set up the delay loop. The DCX B instruction at 0306H decrements the register pair by one. This instruction treats the contents of the B/C pair as a four hex digit number so that both registers are decremented together. In this case only the C register contains data so that is the only register that is actually decremented. Had the B register contained a 01H, the contents of the pair would be decremented as follows:

B register	C register
01	4F
01	4E
01	4D
.	.
.	.
01	02
01	01
01	00
00	FF
00	FE
00	FD
.	.
.	.
00	01
00	00
FF	FF
FF	FE

Notice that when the register pair contains 0000H, the decrementing process continues by bringing up FFFFH. By pressing the STEP key, we can see the timing constant decremented by one.

Enter:	See Displayed:
STEP	⊢0307-79
DCR NXT NXT	ub----00
NXT	uC----4E

The decrementing process has occured as we expected. However, as we have already seen, this process will continue right past zero to FFFFH unless we do something to stop it. Our program must somehow check to see if the B/C register pair contains zero after the decrementing process. If it does, we must exit the loop for the desired time delay has been completed.

While the DCX instruction shortens our program by handling both of the registers at the same time, it does have a minor drawback, it does not set the zero flag. That means that just placing a JNZ instruction after the DCX to set up the loop will not work since the CPU has not set the zero flag when the DCX causes both B and C to contain zero.

We have a trick for quickly detemining whether two registers both contain zero or not and setting the zero flag appropriately. However, the trick only works if the accumulator is one of the registers, so first we must move the contents of one of the registers to the accumulator. The MOV A, C at location 0307H accomplishes that; a MOV A, B would have worked just as well.

Enter:	See Displayed:
STEP	t0308-b0
DCR	uA----4E
NXT NXT	ub----00

Now the two registers in question are the accumulator and the B register. We have an instruction, the ORA R, that checks to see if the accumulator and one other register, R, both contain zeroes. Thus the ORA B instruction at location 0308H sets the zero flag if and only if both the accumulator and the B register contain zero. The contents of the two registers are themselves unchanged by the instruction, only the flag is set depending upon the outcome of the test. Watch!

Enter:	See Displayed:
STEP	t0309-C2
DCR	uA----4E

Well, the accumulator wasn't changed by the ORA B. How about the flags?

NXT	uF----3E
NXT	ub----00

Obviously, the flags have been changed. We don't have enough information about the flag register to interpret its contents yet, but we can tell whether or not the JNZ at 0309H occurs.

Enter:	See Displayed:
STEP	⊢0306-0b

Obviously, the zero flag was not set by ORA B because the JNZ instruction caused a jump back to 0306H. The combination of ORA B and the JNZ set up the loop. We can intervene to make the loop fail by reducing the contents of C to 01H. The effect will be as though the loop had already occured 4DH times and is just about to be concluded.

Enter:	See Displayed:
DCR NXT NXT NXT	uC----4E
0 1	uC----01
NXT	ud----dd

All right, that sets up the C register. Now let's STEP through the remainder of the program and watch it fall out of the loop. The computer is about to execute the DCX B at location 0306H.

Enter:	See Displayed:
STEP	⊢0307-79
DCR NXT NXT NXT	uC----00

If the ORA B and JNZ test is going to fail, it should happen now since the registers both contain zero. STEP twice to execute the MOV A, C and the ORA B.

Enter:	See Displayed:
STEP STEP	⊢0309-C2
DCR STEP	uF----7E

The flags have changed from 3EH to 7EH, and as you will see in Chapter 9, that represents a single flag changing, the zero flag! STEP the next instruction and prove it to yourself. If the zero flag was set, the JNZ will fail and program execution will fall out of the loop and continue at 030CH. If, however, the zero flag was not set the loop will continue and program execution will return to 0306H. Let's find out.

Enter:	See Displayed:
STEP	⊢030C-F1

Obviously our claims about the ORA B were correct; the zero flag was set and the program dropped out of the loop. The next two instructions restore the accumulator, flags and the B/C registers with the values they had before the delay subroutine was CALLed. You will remember that we loaded all of the registers with easy to track data, AA in the accumulator, FF in the flags, BB in the B register, etc.

Enter:	See Displayed:
STEP STEP	⊢030E-23
DCR	uA----AA

NXT	uF----FF
NXT	ub----bb
NXT	uC----CC
NXT	ud----dd
NXT	uE----EE
NXT	uL----04
NXT	uH----01
NXT	uSP-00FE

At this point we have restored all of the registers except the H and L registers. They still contain the address of the least two significant hex digits of the constant in the CALLing program. The delay constant was located at 0103H and 0104H, and they still contain that value. We could reload them with another XTHL to reverse the mischief of our opening of the DELAY subroutine, but then the return address saved in the stack memory would be 0104H and that is certainly not where we wish the return to go. We want 0105H to be the return address, since that is the location of the next instruction in the CALLing program. We can fix the situation very easily right now (or with much difficulty later) by merely incrementing the H/L pair with another INX H. XTHL will then restore the prior contents of the H and L registers and at the same time get the corrected return address into the stack. A RET instruction completes the subroutine.

Enter:	See Displayed:
STEP	t030F-E3

DCR NXT NXT NXT NXT NXT NXT	uL----05
NXT	uH----01

The incrementing of the H/L pair has taken place. Now to get that address back into the stack and restore the H/L pair to their original values.

STEP	⊢0310-C9
DCR	uA----AA
NXT	uF----FF
NXT	ub----bb
NXT	uC----CC
NXT	ud----dd
NXT	uE----EE
NXT	uL----11
NXT	uH----22
NXT	uSP-00FE

There. All of the registers have been restored. All that remains is to execute a RET instruction which will get the return address from the stack. It will also correct the stack pointer itself so that it again contains 0100H as it did before the DELAY subroutine was CALLed. Notice that we could not have used a POP H instruction to restore the H and L registers since that would have screwed up the stack pointer and would not have gotten the corrected return address back into the stack. As it is now we can use DCM to check the stack and prove that the return address is really there.

Enter:	See Displayed:
DCM 0 0 F E NXT	n00FE-05
NXT	n00FF-01

There it is, just waiting for a RET instruction. Let's give it one.

STEP	⊢0105-FF

We're back at the CALLing program, and what's more, we're at exactly the place in that program we should be. We have successfully caused the computer to return to a point in that program two locations beyond where it would normally have returned. All of the registers have been saved and if you check the stack pointer using the DCR mode, you'll find that it is back to 0100H.

Writing the Master Program. Now that we've covered the DELAY subroutine, let's get to work on the big program. We promised to treat this program as an example of how a working programmer might write an actual program. One of the tricks to getting a program up and running fast, is to write a master program that is made up entirely of jump and CALLs. The remainder of the program segments sprinkled around in memory. These are CALLed by the master program, one after another. Since there is unused memory separating the various segments and subroutines, the programmer is able to start with abbreviated versions, test them, expand them, and in doing, develop the entire program with a great deal of confidence that it is going to work correctly when it is completed.

To do this requires that the programmer have a very good idea of the overall program flow that is going to be required before he writes his first line of program. In the case of the programmable controller program that we are working on, that flow will be something like this.

The program will be broken into modules. The first module, called INITIALIZE, sets up the various registers and data memory locations. This is entered automatically when EXC is pressed. This is followed by a module that checks the DIP switches to determine whether the clock keeping portion of the program should be slowed down or speeded up. This module, called DIP This program module is then exited and another, ONE SECOND, is entered. That module uses the DELAY subroutine that we just wrote and debugged as the first of our utility programs. In any event, ONE SECOND causes the displays to light for exactly one second. The information displayed is, of course, the time. At the end of ONE SECOND, another module, UPDATE TIME, is entered. That module increments the memory locations being used to store the current time. Then UPDATE DISPLAYS module loads this information into the display registers so that the next ONE SECOND module will display the corrected time. The information loaded into the display registers by UPDATE DISPLAYS is correct for a twenty-four hour clock. We want this clock to show the time in the more normal 12 hour base but we want an additional feature—the decimal point in the hour display should tell whether the time is AM or PM. All of this is done by a program module called CORRECT DISPLAYS. It operates strictly on the data in the display registers; the data contained in the current time memory locations is unchanged and therefore remains in the twenty-four hour format. Finally the contents of the current time memory locations is compared against all of the preprogrammed times loaded by the user into the memory. Any time a match is found, meaning the computer should perform an action at one of the I/O ports since the current time matches the one preprogrammed by the user. Both time and action are set by the user. This module, REVIEW STACK ENTRY, will be explained more fully in a moment. The overall program flow is shown in Fig. 7-3.

Now that the overall program flow has been established, we can go ahead and write the master program using JMP instructions. Some master programs will be made up entirely of CALLs; others will utilize both JMPs and CALLs. We are going to write an abbreviated version of the master program right now, and then modify it as we add more of the modules. Since we know in advance that this part of the program will be modified we can make life simpler so that new JMPs can be added as we wish without having to key in the entire program each time. The trick lies in the use of the NOP instruction, No Operation. This instruction does absolutely nothing but take up memory space, one location per NOP. When it is executed, nothing changes but the program counter. Even the flags remain the same. By using a combination of JMPs and NOPs we can include the modules we want. Later, to add a new module to the program just replace three of the NOPs with a JMP to that module. Load the following program into the computer.

MASTER PROGRAM

```
0100    C3    JMP    0120    INITIALIZE          :Go to the module to
0101    20                                       ;initialize memory locations.
0102    01                                       ;
0103    00    NOP                                ;There NOPs that will later be
0104    00    NOP                                ;replaced with a JMP to DIP
0105    00    NOP                                ;SWITCH.
0106    C3    JMP    0140    ONE SECOND          ;Go to ONE SECOND to display
0107    40                                       ;current time for one second.
0108    01                                       :
```

```
0109    C3    JMP    0200    UPDATE TIME         ;Go to update to increment the
010A    00                                       ;current time memory locations.
010B    02                                       ;
010C    C3    JMP    0250    UPDATE DISPLAYS     ;Go to update
010D    50                                       ;display module to load display
010E    02                                       ;registers with new time.
010F    00    NOP                                ;Three NOPs that will later be
0110    00    NOP                                ;replaced with a JMP to CORRECT
0111    00    NOP                                ;DISPLAYS.
0112    00    NOP                                ;Three NOPs that will later be
0113    00    NOP                                ;replaced with a JMP to REVIEW
0114    00    NOP                                ;STACK ENTRY.
0115    C3    JMP    0103                        ;Return to 0103H to set up loop.
0116    03                                       ;
0117    01                                       ;
```

Notice that in the master program we have set up the other program modules as being spread out in the system memory. Since we haven't written those modules yet, we don't know how long they will turn out to be. By leaving plenty of room between the starting addresses of those modules, we can work on them with little fear of one running into the next. If we had tried to pack them tightly together in memory, with one starting at the very next location following the final instruction of the preceding mdoule, every time we add an instruction to one of the modules the entire program after that point will have to be rekeyed into the system. Of course, when the entire program has been debugged we may want to rewrite it to pack it into the memory eliminating unused locations separating the modules. But for now, we just want to get the program working as soon as possible and those spaces between the modules are going to save us a lot of time.

One Second. The display subroutine that we developed in the last chapter can be used again. That subroutine utilized an earlier version of DELAY in which the delay constant was perman-

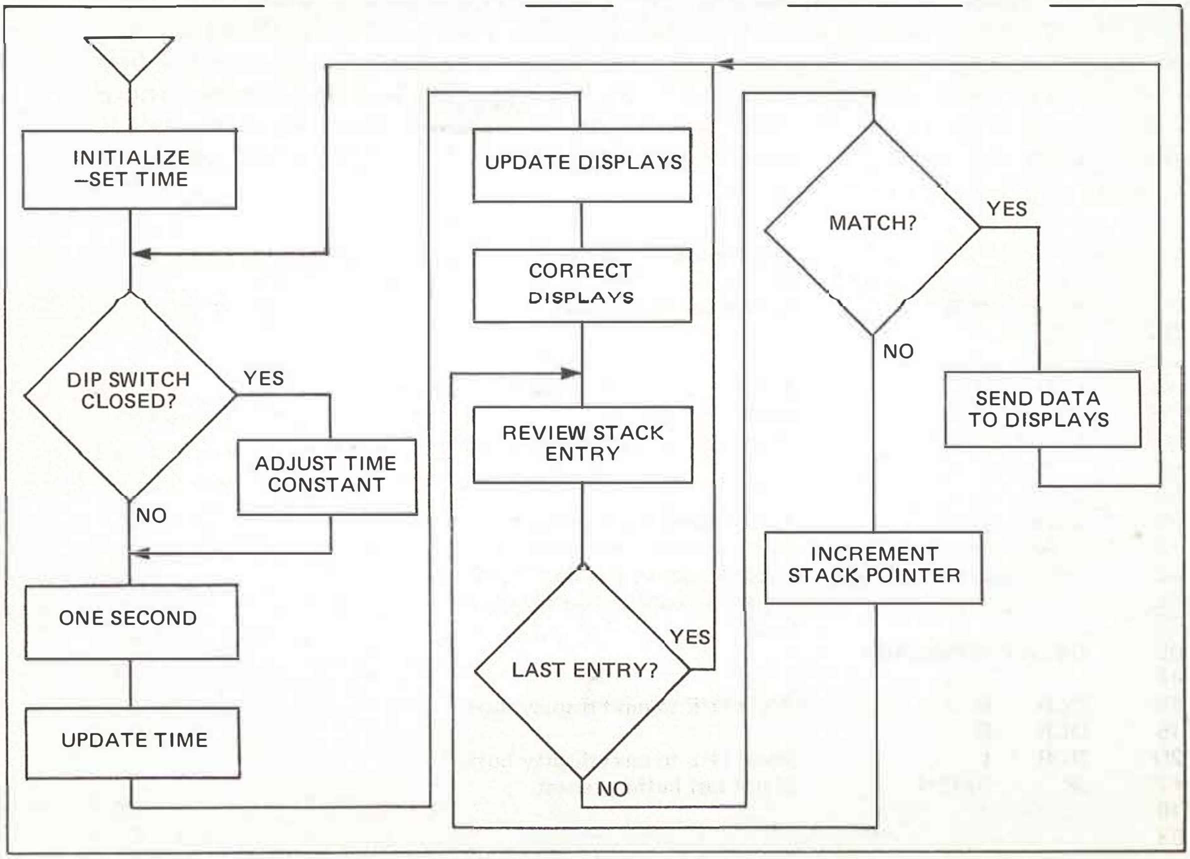


Fig. 7-3 Overall program flow of the programmable controller.

ently set at 4FH and could not be changed. With our new version of DELAY at 0300H we need to include the desired delay constant when we CALL DELAY. Also the original DISPLAY subroutine was located at 0309H, but our new version of DELAY is slightly longer than the first version, so we'll have to move DISPLAY up higher in the memory. The first open location above DELAY is 0311H and that will be the starting location for ~~DELAY~~ DISPLAY. Since the subroutine is almost identical to the one presented in chapter 6 and illustrated in Fig. 6-13, it will not be discussed here. Load DISPLAY into the computer.

DISPLAY ONE-PASS

```
0311    F5    PUSH   PSW            ;Save working registers.
0312    D5    PUSH   D              ;
0313    E5    PUSH   H              ;
0314    16    MVI D, CFH            ;Point D/E to most significant display
0315    CF                          ;digit, CFXXH.
0316    21    LXI H, 0007H          ;Point H/L to display buffer No. 7.
0317    07                          ;
0318    00                          ;
0319    7E    MOV    A,M            ;Load accumulator with contents of
031A    12    STAX   D              ;display buffer. Send contents of
031B    CD    CALL DELAY (0300H)    ;accumulator to display. Delay to
031C    00                          ;simulate guard circuit timing out.
031D    03                          ;
031E    00    DELAY CONSTANT        ;
031F    4F                          ;
0320    15    DCR    D              ;Point D/E to next display digit.
0321    15    DCR    D              ;
0322    2D    DCR    L              ;Point H/L to next display buffer.
0323    F2    JP     0319H          ;If not last buffer, repeat.
0324    19                          ;
0325    03                          ;
```

```
0326    E1    POP    H          ;Restore working registers.
0327    D1    POP    D          ;
0328    F1    POP    PSW        ;
0329    C9    RET               ;Return to calling program.
```

Consulting our master program we find that the ONE SECOND program module is to be located at location 0140H. Now that DELAY and DISPLAY ONE-PASS are stored in the utility portion of the memory, we should be able to come up with a program to drive the displays for one second. In fact, why don't you go ahead and try writing it yourself. Remember to save the registers in the stack before using them. That way we can move from module to module without worrying about which registers have been tied up in the process. Hint: you'll only need one register's worth of looping through the DISPLAY ONE-PASS to come up with one second's worth of display. When you're done, turn the page.

How did you do? Our version is as follows:

```
                                   ONE SECOND

0140    C5     PUSH   B               ;Save registers in stack.
0141    06     MVI B,A0H              ;Load B with loop constant.
0142    A0                            ;
0143    CD     CALL DISPLAY (0311)    ;Display contents of display registers.
0144    11                            ;
0145    03                            ;
0146    05     DCR B                  ;Decrement loop counter.  If zero,
0147    C2     JNZ 0143               ;jump back for another pass.  If
0148    43                            ;not, fall out of loop.
0149    01                            ;
014A    C1     POP B                  ;Restore registers.
014B    C3     JMP 0109H              ;Go back to master program.
014C    09                            ;
014D    01                            ;
```

We really don't know whether the loop constant, A0H, we used in the program is the correct value or not. But remember we're planning to add a module that uses the DIP switches to speed up or slow down the clock. That module will operate by changing the value of the loop constant at 0142H under the control of the DIP switches. Notice that we could have written the final jump address as 0200H instead of 0109H. After all, when the program executes JMP 0109H it is going to find another jump at that point, JMP 0200H, so why not eliminate the extra step and JMP right at the conclusion of ONE-SECOND. If we were worried about saving every possible memory location, we would certainly do that. But right now, we're just trying to get the program working. By using the master program made up entirely of JMP instructions, we can modify the modules easily without affecting one another.

We need to fill in some of the other modules in order to check the programs we have just written. Naturally we can't try to load the entire full-blown modules because we'll never be able to track the program problems down to which module is causing the trouble. Actually, a more likely possibility is that all of the modules will have bugs and we'll never be able to trouble shoot any of them because there will be no display and therefore no visible symptoms. Instead, we should place just enough instructions in each of the other module positions to verify the overall program flow and the correctness of the module we are spending our time on. We need to load instructions into three more of the modules to get the program working. Naturally we want to load as little as possible into these modules so that we don't introduce errors in them that will keep the program from working, since we then will not know whether the problem is in one of the new abreviated modules or not.

The INITIALIZE module was intended to initialize the registers and memory locations used in the program. We need to decide on which locations are going to be used for data. Although we have a wide variety of potential choices we chose 001DH to store the seconds current time, 001EH to store the minutes time, and 001FH to store the hours current time. These locations were chosen because they are immediately above the memory locations used by the monitor program. This keeps them out of the way of the program we're writing. Let's load a program segment to set 001DH, the ~~hours~~ seconds current time location to 00H. Load the following program segment.

INITIALIZE

```
0120    21    LXI H, 001DH    ;Point H/L to current seconds location.
0121    1D                    ;
0122    00                    ;
0123    36    MVI M, 00H      ;Load memory location with 00H.
0124    00                    ;
```

```
0125      C3      JMP 0103H               ;Go back to master program.
0126      03                              ;
0127      01                              ;
```

That is too short to have an error. The INITIALIZE module is the first one executed by the program. Although this version is far from the final one, it will serve to set at least one of the current time locations to 00H so we can see our display modules operate.

After INITIALIZE, the computer will execute ONE SECOND ~~DISPLAY ONE PASS~~ at 0140H. If you have not loaded that module, do so now. The next module is UPDATE TIME at 0200H. It serves to increment the three current time memory locations after every second. For now, let's just increment the seconds memory location at 001DH.

UPDATE TIME

```
0200      21      LXI H, 001DH            ;Point H/L to seconds current time
0201      1D                              ;memory location.
0202      00                              ;
0203      34      INR M                   ;Increment seconds memory location.
0204      C3      JMP 010CH               ;Go back to master program.
0205      0C                              ;
0206      01                              ;
```

This preliminary version of UPDATE TIME could be made shorter yet since H/L are still pointed to 001DH from INITIALIZE. Using another LXI H instruction here is redundant but safe. The chances of these pointers remaining locked together as we expand and develop the modules is small. At some point, we'll add an instruction to CORRECT DISPLAYS and suddenly another module, say UPDATE TIME will go south. The pointers just fell out of step, but you may spend hours searching for it. Life is going to be much simpler if you think of the modules as independent stand-alone programs, at least at this early stage of development.

Of course, without getting the updated contents of the seconds current time memory location in the display register, DISPLAY ONE-PASS will not display the time. The function of the module at 0250H, UPDATE DISPLAYS, is to transfer the data in the current memory locations into the display registers. Since we're only working with the seconds current time at this point, we'll just load it into one of the registers. We could probably set up all of the display registers at this point, but why chance it.

UPDATE DISPLAYS

```
0250    11    LXI D,  001DH          ;Point D/E to seconds current time memory
0251    1D                           ;location.
0252    00                           ;
0253    1A    LDAX D                 ;Load accumulator with seconds current
0254    32    STA 0008H              ;time.  Then load it in right-
0255    08                           ;most display register.
0256    00                           ;
0257    CD    CALL CONVERT (8132H);Convert the contents of the display
0258    32                           ;registers into 7-segment code and
0259    81                           ;load into the diplay buffers.
025A    C3    JMP 010FH              ;Go back to master program.
025B    0F                           ;
025C    01                           ;
```

We are now ready to try our first pass at the program.

Enter:	See Displayed:
CLR EXC	n - - - - - - 1

The displays will remain in the DCM mode for about one second after you press EXC, then change to 1 for another second, 2 for the next, and so on. Apparently our programs are operating so far. In particular, the DISPLAY ONE-PASS subroutine appears to be correct.

The program needs some way to set the correct time into the current time memory locations so the computer can be set to the correct time of day. That would best be done by INITIALIZE so let's go back and extend that module a little. We'll use the B and C registers accessed through DCR mode to enter the correct time. The assumption is made here that the seconds are always set at 00H by the computer. The user sets the correct hours into the B register and the correct minutes into the C register. The INITIALIZE module then loads these into the appropriate current time memory locations.

INITIALIZE

```
0120    21    LXI    H, 001D    ;Point H/L to seconds current time
0121    1D                      ;memory location.
0122    00                      ;
0123    36    MVI    M, 00H     ;Load seconds current time with 00H.
0124    00                      ;
0125    23    INX    H          ;Point to minutes time location.
0126    71    MOV    M, C       ;Load user's minutes into minutes
0127    23    INX    H          ;memory location.  Point to hours
0128    70    MOV    M, B       ;time location.  Load user's hours
0129    C3    JMP    0103H      ;into hours memory location.  Go
012A    03                      ;back to master program.
012B    01                      ;
```

This will allow the user to load the correct time into the current memory locations via the DCR mode. We still need some way to get the data in those locations into the displays. Before we can try this new version of INITIALIZE we will have to expand UPDATE DISPLAYS so the hours and minutes are shown as well. Add the following program segment to that already in memory.

UPDATE DISPLAYS cont.

```
0257    13    INX    D            ;Point D/E to minutes memory location.
0258    1A    LDAX   D            ;Load accum with current minutes.
0259    32    STA    0009H        ;Store current minutes in next display
025A    09                        ;register.
025B    00                        ;
025C    13    INX    D            ;Point D/E to hours memory location.
025D    1A    LDAX   D            ;Load accum with current hours.
025E    32    STA    000AH        ;Store current hours in next display
025F    0A                        ;register.
0260    00                        ;
0261    CD    CALL CONVERT (8132H);Convert display registers into 7-
0262    32                        ;segment code in display buffers.
0263    81                        ;
0264    C3    JMP 010FH           ;Go back to master program.
0265    0F                        ;
0266    01                        ;
```

To test these expanded modules we need to load the B and C registers with the desired time and then press EXC . Our program can only handle one-digit numbers at this time, so keep it simple. Assume the time is 1:05.

Enter:	See Displayed:
DCR NXT NXT 0 1	ub----01
NXT 0 5	uC----05
NXT CLR EXC	n----151
	n----152
	n----153
	:

That gets the user's time into the system all right, but the rest of the displays are getting confusing. They still contain the remnants of the DCM mode displays. We could include code in the INITIALIZE module to go through and blank the rest of the displays individually, but there's a much easier way. In the monitor program at location 80CCH is a CALLable subroutine used to blank all of the displays. If you perform a CALL to that address before leaving INITIALIZE, the rest of the useful monitor subroutines are noted on the bottom of the Hex Card at the front of this binder. Notice, for instance, that CONVERT is listed there, along with its address, 8132H. There too, is DISPLAY-BLANK 80CCH.

INITIALIZE cont.

```
0129    CD      CALL BLANK (80CCH)  ;Blank all displays.
012A    CC                          ;
012B    80                          ;
```

```
012C    C3    JMP    0103H      ;Go back to master program.
012D    03                      ;
012E    01                      ;
```

Enter: CLR EXC

See Displayed:

```
      151
      152
      153
```

Well, we're getting closer, but it still doesn't look much like a clock. For one thing, a digital clock is made up of three 2-digit groups, one for hours, one for minutes and one for seconds. Furthermore, the groups should be separated by spaces for readability. Since all the displays were blanked by the subroutine in INITIALIZE, this could be done by just incrementing the H/L pointer past the displays to be left blank. There is another problem, though. Each of the memory locations contains two hex digits. These digits can express a range sufficient for that of the seconds or minutes, 60, and certainly enough for hours. But so far, we have always moved both digits into a display register. Of course, we had to be careful that the first digit was always zero, because the LED can only handle one digit at a time in the display. We got around this by making sure that our display registers and buffers never received a number that did not have a zero as its first digit. In this program, the memory locations do have two digits and we cannot arbitrarily drop one of them. Somehow we must get the first digit into one of the dispalys and the other digit into another, adjacent display. It turns out that this problem is so common there is a subroutine in the monitor just for it. It is called SUB3 and is located at 80A5H. Since this subroutine is not noted on the bottom of the hex card you may want to write that location down.

To use SUB3 point H/L to the most significant of a display register pair, load the C register with 02H, and then CALL SUB3. The subroutine will take the contents of the accumulator and send the least significant of the hex digits to the least significant of the display register pair, and the most significant hex digit to the most significant of the display register pair. For instance if the accumulator contains A3H, H/L are pointing at 0009H and C contains 02H, CALLing SUB3 will then load the display register at 0009H with 0AH and the register at 000AH with 03H. Both H/L and C will be unaffected by the CALL.

Let's put this to work. In our program the seconds current time register at 001DH contains a two hex digit number that represents the seconds portion of the clock display. These digits should appear in the two right-most displays as the seconds portion of the clock display. To achieve that the contents of the seconds current time memory location must be loaded into the two display registers at 0009H and 0008H. Load the following program into the computer.

UPDATE DISPLAYS

```
0250    11      LXI D,  001DH           ;Point D/E to seconds current time
0251    1D                              ;memory location.
0252    00                              ;
0253    21      LXI H,  0009H           ;Point H/L to most significant of
0254    09                              ;display register pair.
0255    00                              ;
0256    01      LXI B,  0002H           ;Load B with 00H and C with 02H.
0257    02                              ;
0258    00                              ;
0259    1A      LDAX    D               ;
025A    CD      CALL SUB3 (80A5)        ;Load display register pair with
025B    A5                              ;accum contents—current seconds.
025C    80                              ;
025D    13      INX     D               ;Point D/E to minutes current time
```

```
025E     23     INX     H              ;memory location.  Point H/L to space
025F     09     DAD     B              ;between seconds and minutes.  Increment
0260     1A     LDAX    D              ;H/L by 2 so it points to most significant
0261     CD     CALL SUB3(80A5)        ;of display register pair.  Load accum
0262     A5                            ;with current minutes.  Load display
0263     80                            ;register pair with current minutes.
0264     13     INX     D              ;Point D/E to hours current time
0265     23     INX     H              ;memory location.  Point H/L to space
0266     09     DAD     B              ;between minutes and hours.  Increment
0267     1A     LDAX    D              ;H/L by 2 so it points to most signi-
0268     CD     CALL SUB3 (80A5)       ;ficant of display register pair.  Load
0269     A5                            ;accum with current hours.  Load display
026A     80                            ;register pair with current hours.
026B     CD     CALL CONVERT (8132)    ;Convert contents of display registers
026C     32                            ;into seven segment code and load into
026D     81                            ;display buffers.
026E     C3     JMP     010FH          ;Go back to master program.
026F     0F                            ;
0270     01                            ;
```

The program operates by pointing H/L to the most significant of the seconds display register, 0009H, and the D/E pair to the memory location holding the current count of the seconds, 001DH. We load B with 00H and C with 02H with an LXI B instruction. Even though the B and C registers are used for different tasks in this subroutine, we can save program space by loading them both at the same time. The LDAX D instruction loads the accumulator with two hex digits that represent the current seconds count. Everything is now set up for CALLing SUB3. H/L are pointed to the dipslay registers, C has 02 in it and the accumulator has been loaded with the correct data. When CALL SUB3 is executed, the contents of the accumulator will be loaded into the display registers. We can then increment D/E so they point to the minutes current memory location, 001EH. An INX H instruction increments the H/L pair so it points to the display register that corresponds to the space between the seconds and the minutes displays. To use SUB3 again, we have to increment H/L by another two so that it points to the most significant of the dispaly register pair that will contain the minutes display, 000CH. Of course we could do that by using two more INX H instructions, but there is an easier way. Since B/C contain 0002H, adding the B/C register pair to the H/L register pair will have the effect of incrementing it by 2. Normally this would be far more involved than just using two INX H instructions, but it happens that there is an instruction just for this purpose. DAD B, Double Add B/C, acts to add the four digit contents of the B and C registers to the four digit contents of the H and L registers. The results are stored in the H and L registers. Between the single INX H and the DAD B, we have managed to increment the H/L pair by three so that it now points to the most significant of the display register pair that will contain the minutes display. Load the accumulator with another LDAX D instruction and perform another CALL SUB3 and the minutes displays are loaded. Repeating the process loads the hours displays. All the remains is to CALL CONVERT in order to convert the contents of the display registers into seven segment code in the display buffers.

Enter:	See Displayed:
DCR NXT NXT 1 0	ub----10
NXT	uC----46
CLR EXC	10 46 01

Now we're starting to get close. Hours, minutes and seconds are all being displayed in the proper places. Of course, only the seconds are counting and they're doing it in their own kind of system, but so far we've been making pretty good progress. Actually, if you've been watching the displays for more than a few seconds now, you probably realize that the seconds are counting in the now familiar hexadecimal system. In the last chapter we worked with an effective, if somewhat lengthy, method of making the system count in decimal instead.

Decimal Accumulator Adjust. The need for decimal arithmetic is so common that there is an instruction provided just to simplify it. The instruction, DAA or Decimal Accumulator Adjust, operates on the contents of the accumulator to convert the two hexadecimal digits into two decimal digits.

In order for the DAA instruction to operate correctly it is necessary that the numbers used already be in the decimal format. Why do we need an instruction to convert numbers into decimal if they have to already be in decimal for the instruction to work? Consider this example. If you add 7 to 24 with paper and pencil, you'll get 31. If you add 7 to 24 with the ia7301, you'll get 2BH. The DAA instruction operates on the results of arithmetic operations on decimal numbers, and it serves to convert those results which are in hexadecimal into straight decimal. The key to the operation of the DAA instruction is the fact that there are six more numerals in the hexadecimal number system than in the decimal system. Therefore to convert a hex digit to decimal, just add six if the hex digit is greater than nine. If it is nine or less, no action is required. Thus 7H is the same as decimal 7, since 7H is 9 or less. But BH is greater than 9 so we add 6 to it and get 11.

$$\begin{array}{r} \text{BH} \\ \underline{+6\text{H}} \\ 11\text{H} \end{array}$$

The DAA instruction utilizes two flags associated with operations in the CPU. The first of these is the Carry flag. It is set every time an operation results in an overflow. Notice that the operation need not involve the accumulator. Those instructions that set or reset the carry flag are specified on the Hex Card, in fact, all of the instructions are specified relative to the flags they affect. We are already familiar with the zero flag that is set when certain operations result in the contents of a register or memory location becoming zero. Keep in mind that it takes an operation to set the flags. Merely moving data into a register will not set any of the flags; we have to add, subtract, increment or perform some other operation to set the flags. If we do not wish to change the data itself, but still need to know whether it is zero or not, we can add zero to it. The addition operation sets the flags, but the fact that the operand is zero means the data itself is unchanged.

When an operation results in an overflow from a register or memory location, the carry flag will be set. An exception to this is the INR R or DCR R instructions which set all of the flags EXCEPT the carry flag. That is to facilitate certain arithmetic operations by making it possible to preserve the status of the carry flag even though the register contents are incremented or decremented. In particular the carry flag is set during addition and subraction operations. The carry flag is only set if an overflow out of the entire register results; it is not set during operations which cause overflows out of the first hex digit of the contents. Only those operations that cause an overflow out of the second, or most significant, of the hex digits will set the carry flag. To make use of the information contained in the carry flag there are a set of conditional jumps, calls and returns that only occur depending upon the status of the carry flag.

In addition to the carry flag, there is a second flag, called the Auxilliary carry that is set or reset depending upon whether or not there is an overflow from the first, or least significant digit. Unlike the carry flag, there are no conditional jumps, calls or returns that depend upon the status of the auxilliary carry. In fact, there is one and only one instruction in the entire instruction set that depends upon that flag. The instruction is the DAA instruction, and it uses both the carry and the auxilliary carry to performits hexadecimal to decimal conversion. We will first state the rules by which DAA operates, and then illustrate that operation with an example or two.

Rule 1. If the least significant hex digit of the accumulator is a number greater than 9, or if the auxilliary carry flag is set, the accumulator is incremented by 6. Otherwise, no incrementing occurs.

Rule 2. If the most significant hex digit of the accumulator now is a number greater than 9, or if the carry bit is set, the most significant hex digit of the accumulator is incremented by 6. Otherwise, no incrementing occurs.

Rule 3. If a carry out from the least significant hex digits occurs during the operation in Rule 1, the auxilliary carry flag is set; otherwise it is reset. Likewise, if there is a carry out of the most significant hex digits during the operations described by Rule 2, the normal carry flag is set; otherwise, it is unaffected.

With these three rules in mind, let's try an example. Assume the accumulator contains 24H and we add 07H to it. After the

$$\begin{array}{r} \text{24H} \\ \text{+07H} \\ \hline \text{2BH} \end{array}$$

addition the accumulator will contain 2BH. If we now perform a DAA instruction, the results will be as follows. By Rule 1, the least significant hex digit, B, is greater than 9 so we must add 6 to it. That results in a carry out from the

$$\begin{array}{r} \text{24H} \\ \text{+07H} \\ \hline \text{2BH} \\ \text{+06H} \\ \hline \text{31H} \end{array}$$

least significant hex digit, so the auxilliary carry is set. This carry is added into the addition of the most significant hex digits so that the 2 in the most significant digits place is incremented to 3. The result, 31, is the correct decimal number that should result from adding decimal 24 and decimal 07 together.

For a second example, suppose the accumulator contains 24 and we add 87 to it. In hexadecimal the result, stored in the accumulator, will be ABH. That is,

$$\begin{array}{r} 24\text{H} \\ +87\text{H} \\ \hline \text{ABH} \end{array}$$

If we were performing this arithmetic operation as part of some sort of decimal routine, where the numbers 24 and 87 were being considered as decimal numbers, it would be necessary to generate a decimal result. But because the computer expresses its numbers in hexadecimal, thinking of the two input numbers as decimal quantities does not stop the computer from thinking of them as hexadecimal and generating the correct hexadecimal result. The answer, of course, is to perform a DAA on the result in the accumulator to get it into a hexadecimal number that appears as decimal. In the case of the least significant digit, the numeral B is larger than 9 so 6 is added to the accumulator. This results in a carry out so the auxilliary carry is set.

$$\begin{array}{rr} & 24\text{H} \\ & +87\text{H} \\ \cline{2-2} & \text{ABH} \\ & +06\text{H} \\ \cline{2-2} & \text{B1H} \\ & +60\text{H} \\ \cline{2-2} \text{CARRY OUT} \rightarrow & 1\ 11\text{H} \end{array}$$

The addition of 06H to the accumulator results in it now containing B1H. The most significant hex digit is now examined and, since it is larger than 9, 6 is added to it. This is shown as the addition of 60H to the entire contents. The result is that the accumulator now contains 11H and the carry flag is set. This in effect means the accumulator now contains 1 11 which is the correct result if decimal 24 and decimal 87 are added together.

We can incorporate the DAA instruction into our programmable controller program by placing it in the UPDATE TIME module. That program segment presently operates by pointing H/L to the seconds current time memory location and performing an INR M to increment the count. The result of that program segment is that the seconds count all right, but they count in hexadecimal. In order to use the DAA instruction to make the seconds count in decimal numbers, it will be necessary to bring the current count into the accumulator since the DAA instruction only works on that one register. After the incrementing has taken place, a DAA will be performed to correct the result and then the result moved back to the current time memory location.

One of the very few places where the CPU's used in the ia7301 differ from one another is in the way the DAA instructions is performed. The problem arises from the fact that some of the CPUs set the auxilliary carry flag after INR R instruction. Since the count is going to be brought to the accumulator because of the need for the DAA instruction, we have at hand a wider variety of instructions from which to choose. In particular, we can use an ADI instruction, Add Immediate, to increment the contents of the accumulator. This instruction adds the immediate data that accompanies it to the contents of the accumulator. In the case of the ADI 01H instruction the effect will be the same as the INR except that both types of CPUs do correctly set the auxilliary carry flag with ADI whereas some do not set the flag with INR A. Load the following program into the computer.

UPDATE TIME

```
0200    21    LXI H,  001DH    ;Point H/L to the seconds current time
0201    1D                     ;memory location.
0202    00                     ;
0203    7E    MOV     A,M      ;Bring current seconds to accumulator.
0204    C6    ADI     01H      ;Increment current seconds count.
0205    01                     ;
0206    27    DAA              ;Correct count to decimal number.
0207    77    MOV M, A         ;Load new value back into current
0208    C3    JMP     010CH    ;time memory location.
0209    0C                     ;Go back to master program.
020A    01                     ;
```

When you press CLR and EXC now, you'll see the displays operate as before, except now the seconds are counting by tens instead of 16's. The DAA instruction makes short work of converting what is basically a hexadecimal computer into a decimal version. When the seconds count reaches 09, the next second makes it 10 instead of 0A. And when the count reaches 99, the next second brings 00, not 9A.

We can extend this concept to make the hours and minutes count in decimal numerals, too. So far we have only been counting seconds while we worked out the question of dissplays, timing loops, etc. Now, let's make the hours and minutes functional as well. The following program segment is a simple extension of the one we just tried. Load it into the computer.

UPDATE TIME

```
0200    21    LXI H,  001DH    ;Point H/L to seconds current time
0201    1D                     ;memory location.
0202    00                     ;
0203    01    LXI B,  6024H    ;Load B with 60H and C with 24H.
0204    24                     ;
0205    60                     ;
0206    7E    MOV     A,M      ;Fetch current seconds to accum.
0207    C6    ADI     01H      ;Increment current seconds count.
0208    01                     ;
0209    27    DAA              ;Adjust current seconds to decimal.
020A    77    MOV     M,A      ;Load new value of current seconds
020B    B8    CMP     B        ;back into memory.  Compare new
020C    C2    JNZ     010CH    ;count with 60.  If no match, go to
020D    0C                     ;master program.  If 60, load 00
020E    01                     ;back into current seconds location.
020F    74    MOV     M,H      ;
0210    23    INX     H        ;Point H/L to minutes current time.
0211    7E    MOV     A,M      ;Fetch current minutes to accum.
0212    C6    ADI     01H      ;Increment current minutes count.
0213    01                     ;
0214    27    DAA              ;Adjust current minutes to decimal.
0215    77    MOV     M,A      ;Load new value of current minutes
0216    B8    CMP     B        ;back into memory.  Compare new
0217    C2    JNZ     010CH    ;count with 60.  If no match, go to
0218    0C                     ;master program.  If 60, load 00
0219    01                     ;back into current minutes location.
021A    74    MOV     M,H      ;
021B    23    INX     H        ;Point H/L to hours current time.
021C    7E    MOV     A,M      ;Fetch current hours to accum.
021D    C6    ADI     01H      ;Increment current hours count.
021E    01                     ;
```

```
021F      27      DAA                   ;Adjust current hours to decimal.
0220      77      MOV     M,A           ;Load new value of current hours back
                                         into memory.  Compare new
0221      B9      CMP     C             ;count with 24.  If no match, go to
0222      C2      JNZ     010CH         ;master program.  If 24, load 00
0223      0C                            ;back into current hours location.
0224      01                            ;
0225      74      MOV     M,H           ;
0226      C3      JMP     010CH         ;Go back to master program.
0227      0C                            ;
0228      01                            ;
```

This version of UPDATE TIME is simply an extension of the one we have already worked with. H/L are pointed to the seconds current time memory location, this value is brought to the accumulator, incremented by one, adjusted to a decimal number, and then reloaded into the memory location from whence it came. Here we depart from our first version. That program segment allowed the clock to run freely, never going back to 00 after the display comes up 59. Now when we reach that point we check to see if the incremented value of the seconds count is 60. We do that by comparing the contents of the accumulator, the current seconds count, to the contents of the B register, 60H. The CMP B instruction operates in the same way as the CPI instruction we are already familiar with. The only difference is that while the CPI instruction compares the contents of the accumulator with the immediate data supplied by the instruction, the CMP B instructions compares the contents of the accumulator with the contents of the B register. Since that register contains 60H, a match will occur if and only if the current seconds count is 60H. Like the CPI instruction, CMP B sets the zero flag when a match is made. The JNZ instruction that follows the CMP B causes the program execution to return to the master program if the seconds count is anything but 60H. If the count is 60H, however, we need to zero out the seconds memory location and increment the minutes count. We could do this with a MVI M, 00H, but a much simpler way is to just MOV M,H. Since the H register coincidently contains 00H, the effect is the same but requires less program steps. The program is repeated for minutes and hours. When the time comes to check the hours count to see if it contains 24H, the comparison is made against the C register which contains 24H. For this purpose, CMP C is used.

When this version of UPDATE TIME is loaded into the computer and executed, you'll find yourself with a 24 hour clock. The time has come, however, to correct the 23 hour displays to the more common 12 hour type we are familiar with. The module that accomplishes that, called CORRECT DISPLAYS, is located at 0275H. To get there we need to go back to the master program and replace three of the NOPs at 0112H, 0113H and 0114H with a JMP 0275H. That will cause program execution to go to CORRECT DISPLAYS after UPDATE DISPLAYS.

We have three problems with the displays. First, our clock presently displays the hour as 00, even though every other clock I've ever seen displays midnight as 12. Then, to get our clock in the standard 12 hour format, it is necessary to subtract 12 from the time whenever the hours count is greater than 12. Finally, we wish to add a dot to tell us when the time is AM rather than PM. These three problems are all to be handled by the CORRECT DISPLAYS module. The requirements are summarized below:

Hours	Display	AM Dot	Table	Entry
00	12	Yes	02B0H	80H
01	01	Yes	02B1H	8BH
02	02	Yes	02B2H	8BH
03	03	Yes	02B3H	8BH
04	04	Yes	02B4H	8BH
05	05	Yes	02B5H	8BH
06	06	Yes	02B6H	8BH
07	07	Yes	02B7H	8BH

08	08	Yes	02B8H	8BH
09	09	Yes	02B9H	8BH
10	10	Yes	02C0H	8BH
11	11	Yes	02C1H	8BH
12	12	No	02C2H	88H
13	01	No	02C3H	95H
14	02	No	02C4H	95H
15	03	No	02C5H	95H
16	04	No	02C6H	95H
17	05	No	02C7H	95H
18	06	No	02C8H	95H
19	07	No	02C9H	95H
20	08	No	~~02CAH~~ 02D0H	95H
21	09	No	~~02CBH~~ 02D1H	95H
22	10	No	~~02CCH~~ 02D2H	95H
23	11	No	~~02CDH~~ 02D3H	95H

Never mind the columns marked Table and Entry for now; we'll have plenty to say about them in a second. As you examine the chart you will become aware of four possibilities, all involving the AM dot and the hours display. First, at hour 00 the display must be corrected to show 12 and an AM dot added. Then, for hours from 01 to 11 the display remains unchanged, but the AM dot must still be added. At hour 12 the displays are to show 12 but the AM dot is dropped. Finally, from hours 13 to 23, 12 must be subtracted from the hours. There is no AM dot during these hours. It is possible to write a program module made up of conditional jumps that analyzes these possibilities and acts accordingly. We did this just for fun, but the final program, although it works, is rather confusing and not very instructional. For that reason we decided to follow a different tack.

The method that we ultimately used for CORRECT DISPLAYS was to build a table, one location and entry for each of the 23 hours. We then used the contents of the hours current time memory location to access the correct entry in the table. That entry caused a jump to one of four program segments. Each of the program segments dealt with one of the four possibilities just described. Load the program below into your computer. Also load the table in the chart we have just examined into the computer at the addresses indicated. Be careful; some of the entries are close to the same value but a mistake here will generally wipe out your program. Also, there is a discontinuity in the addresses of that table. They jump from 02B9H to 02C0H; the locations in between are unused.

CORRECT DISPLAYS

```
0275    3A     LDA     001FH          ;Load the accum with hours current
0276    1F                            ;time.
0277    00                            ;
0278    21     LXI H,  02B0H          ;Point H/L to first table entry.
0279    B0                            ;
027A    02                            ;
027B    85     ADD     L              ;Add L to hours so accum contains
0270    6F     MOV     L,A            ;least significant two digits of
027D    6E     MOV     L,M            ;table entry.  Move to L and use H/L
027E    E9     PCHL                   ;to fetch jump address to L.  Go
                                      ;to that address.

0280    21     LXI H,  0006H          ;Case I.  Point H/L to display buffer
0281    06                            ;No. 6 and load a seven segment
0282    00                            ;code for a two in it.
0283    36     MVI M, A4H             ;
0284    A4                            ;
0285    23     INX     H              ;Point H/L to buffer No. 7 and load a
0286    36     MVI M, 79H             ;seven segment code for a 1 and a
0287    79                            ;decimal point for the AM dot in it.
0288    C3     JMP     0112           ;Go back to the master program.
0289    12                            ;
```

```
028A   01                                  ;Case II.
028B   21     LXI H,  0007H                ;Point H/L to display buffer No. 7.
028C   07                                  ;
028D   00                                  ;
028E   7E     MOV     A,M                  ;Fetch the seven segment code already
028F   E6     ANI     7FH                  ;in that buffer to the accumulator
0290   7F                                  ;and add the AM dot (decimal point).
0291   77     MOV     M,A                  ;Return the corrected value to the
0292   C3     JMP     0112H                ;display buffer.  Go back to the
0293   12                                  ;master program.
0294   01                                  ;
0295   3A     LDA     001FH                ;Case IV.  Load the accum with the
0296   1F                                  ;current hours count.  Subtract 12
0297   00                                  ;to correct it to PM.
0298   DE     SBI     12H                  ;
0299   12                                  ;
029A   27     DAA                          ;Correct the result to a decimal number.
029B   21     LXI H, 000FH                 ;Point H/L to dipslay register No. 7.
029C   0F                                  ;
029D   00                                  ;
029E   0E     MVI C, 02H                   ;Load C with 02H and then CALL SUB3
029F   02                                  ;to load the results into display
0240   CD     CALL SUB3 (80A5)             ;registers 7 and 6.
02A1   A5                                  ;
02A2   80                                  ;
02A3   CD     CALL CONVERT (8132)          ;Convert hex code in registers into
02A4   32                                  ;seven segment code in the buffers.
02A5   81                                  ;
02A6   C3     JMP     0112H                ;Go back to the master program.
02A7   12                                  ;
02A8   01                                  ;
```

Don't forget to add the correct JMP instruction in the master program; if you don't the CORRECT DISPLAYS module will never be executed. Here is the real value of the master program. It allows us to add or drop modules at will, by changing NOPs to JMPs and back again.

```
010F     C3     JMP     0275H          ;
0110     75                            ;
0111     02                            ;
```

The CORRECT DISPLAYS module is made up of two parts. The first fetches the current hours count from 001FH and uses it to determine the correct address in the table. The table begins at location 02B0H; and LXI H instruction points that register pair to the first entry in the table. We will use the current hours count, now in the accumulator, to modify H/L. This is done with an ADD L which adds the hours count to the least significant digits of the table address. The result is stored in the accumulator, so a MOV L,A instruction is needed to load this portion of the address back into the L register. H/L now contain the address of the table entry that corresponds to the current hours count. MOV L,M fetches the table entry to the L register. When you loaded the table into addresses 02B0H to 02CDH, you were undoubtedly aware of the fact that there were only four different entries in that table. These entries are, in fact, the least significant digits of the address of the program segment written to service that particular case. The entry for hour 00, for instance, is 80H. When 80H is moved to the L register by MOV L,M, it completes the address of the service segment, 0280H, written to service the case when the hours count was 00H. The next task is to somehow perform a jump to that address.

This is easier said than done. With the instuctions already in our repetoire, the only way we could accomplish a jump to the address held in the H/L register pair would be to get that address into the stack memory and perform a RET. Fortunately there is a much easier way since the CPU manufacturer has furnished us with the PCHL instruction. This is another reason why the H/L registers are the most powerful of the working register pairs. The PCHL instruction loads the contents of the H/L registers into the Program Counter. As soon as that is accomplished the computer will continue program execution at the new point just as though a jump had occured, and indeed, it has. The PCHL works like a jump in that the stack memory is not involved. It is very useful for jumping to table entries in exactly the way we just used it.

The PCHL instruction at location 027EH causes program execution to continue at one of four points depending upon the table entry that was fetched during the first part of the module. The rest of the module is made up of three parts, each designed to service one of the four table entries. The disparity between number of table entries and number of service segments is taken care of by the fact that one of the table entries causes a jump to a point near the end of one of the three service segments. One of the segments thus handles two of the table entries.

The first service segment is designed to handle the 00 hour situation. It is located at 0280H. It operates by loading the seven segment code for a 2 into display buffer No. 6 and the code for a .1 into display buffer No. 7. At the conclusion of this segment the display buffers contain the correct code for .12. At the end of this segment a jump takes program execution back to the master program. The table entry at 02B0H is 80H; when that entry is combined with the 02H that is already in the H register, the stage is set for a jump to 0280H which is the entry point for this segment.

For hours between 01 and 11, the second segment at 028BH is called for. All that this segment must do is add the AM dot. It does that by pointing H/L to display buffer No. 7 and fetching the contents of that buffer to the accumulator. There the AM dot is added by the simple expedient of adding a decimal point to the number in the buffer. The process of adding a decimal point is done by the ANI 7FH instruction. Trying to explain this instruction is something of a problem, since it deals directly with the binary number system, something that we will cover in detail in Chapter 9. All we can say about it at this time is that the various registers and memory locations of the computer are all made up of eight individual little storage boxes. The contents of these boxes, called bits, can be either electrically high or low. We have taught the ia7301 to respond to and display its results in the hexadecimal system to simplify the interface to human eye and fingers. But this does not change the basic fact that the computer operates with eight bit words, each bit being either high or low. When we send data to the displays, seven of those bits are used

to determine which of the segments will light to display the hex digit. The eight bit causes the decimal point to light. The ANI instruction causes selective bits to be forced low. ANI 00H would set all of them low. ANI FFH would set none of them low. The ANI 7FH instruction that we use in this program sets one and only one bit low, the one that lights the decimal point. We will have much more to say about this matter in Chapter 9 so please bear with us until then.

Once the ANI 7FH insturction has added the decimal point, all that remains is to send it back to the display buffer with a MOV M,A instruction and then perform a jump back to the master program.

Examining the table of hours and corrected displays again, you will find there is only one hours count that produces the correct displays with no corrections. That count is 12, or high noon. The table entry for that particular count is 88H. If you track that back into the service segments you'll find yourself at 0288H which is a jump back to the master program. This entry, in other words, scoots around the service segments and gets program execution back to the master proprogram without correcting any of the displays.

All of the counts from 13 to 23 represent the PM hours. They produce no AM dot, but do require that 12 be subtracted from the current hours count in order to correct the 24 hour cycle to a 12 hour cycle. That service segment begins at 0295H. It operates by loading the accumulator with the current hours count from 001FH, and then subtracting 12 from that count. The subtraction process is done quite simply with an SBI 12 instruction. That operates by subtracting

12H from the contents of the accumulator and storing the results back in the accumulator. That result is in hexadecimal, so we correct it to decimal by a DAA instruction. To get the result back into the display registers we will use SUB3 again. To do that we have to set up the registers we will use SUB3 again. To do that we have to set up the registers by loading H/L with the address of the most significant of the display register pair and register C with 02H. When that is done we CALL SUB3 to load the display registers, and then CALL COVERT to load the display buffers with the correct seven-segment code corresponding to the numbers in the display registers. Go ahead and execute it; you'll be happy with the results.

Stack Review. At this point we have a very interesting digital clock. All of the problems associated with the clock of the last chapter have been eliminated. But still, we have only a clock. Let's get to the part where we use the clock for controlling things. The ia7301 has an output port with eight separate lines, each of which could be used to control an individual circuit. To demonstrate that capability you would have to build interface circuits and hook them between the computer and the various circuits to be controlled. Since we don't want to wait while you do that, we're going to demonstrate the use of the computer as a controller by using the three discrete LEDs that appear in the left bottom corner of the computer. While this may seem quite a come-down from controlling the world, be assured that the change is a small one. All you need do is go through the program and switch the address of the LEDs, 6000H, to that of the I/O port, 5000H, and the controller will be operating with the output port instead of the LEDs. In the meantime the use of LEDs makes it easy to see the program in action.

The STACK REVIEW module is located at 01A0H. It operates by fetching from memory an action code and a comparison code. If the current time as contained in 001DH, 001EH and 001FH, matches the comparison code, the computer will send the action code to the LEDs. The codes are stored in four memory locations, beginning at 0040H. Here the user stores the action code, followed by the hours comparison code in 0041H, the minutes comparison code in 0042H, and the seconds comparison code in 0043H. All three of the comparison codes must match before the action code can be sent. If the comparison code fails, the computer goes on to the next four memory locations, 0044H through 0047H, and examines the codes there for a match. This continues up to location 00A0H which is the top of this storage area. If no matches are made, the computer goes back to the master program and counts off another second. The search through the comparison code area occurs once every second. That allows the user to load in an action code for any second in the entire twenty-four hour period. When the computer finds a match, it sends the action code to the LEDs and goes back to the master program for another second. While the action codes here are simply instructions for the LEDs to light, they could control other things. Each code contains eight bits of data, each of which could control an external circuit. With the possibility of controlling eight external circuits, all independently in time increments as small as one second, the controller becomes a very real system not to be passed over lightly.

To fetch the comparison codes from the memory in order to compare them to the current time count, we could use the common methods of pointing H/L to the code and then MOVing the codes one by one into the computer. A faster way is to treat the comparison part of the memory as a second, independent stack memory. Data is pulled from the memory by a series of POP instructions that are much quicker and more efficient than MOVs. The program is shown below. Load this into your computer and then we'll discuss it.

REVIEW STACK

```
01A0    31    LXI SP, 0040H       ;Load the stack pointer with first
01A1    40                        ;address of user's
01A2    00                        ;stack.
01A3    21    LXI H, 0000H        ;Load H/L with zero.  Add stack pointer
01A4    00                        ;to H/L so those registers now contain
01A5    00                        ;value of stack pointer.
```

```
01A6    39    DAD    SP          ;This allows us to check to see if SP
01A7    7D    MOV    A,L         ;has reached last entry, 00A0H, yet.
01A8    FE    CPI    A0H         ;If it hasn't, go to 01B3H to review
01A9    A0                       ;the comparison codes against the
01AA    C2    JNZ    01B3H       ;current time count.  If this is
01AB    B3                       ;the top of the user's stack,
01AC    01                       ;fall out of the loop.  Reset the
01AD    31    LXI SP, 0100H      ;stack pointer to that of the master
01AE    00                       ;program and go back there.
01AF    01                       ;
01B0    C3    JMP    0115H       ;
01B1    15                       ;
01B2    01                       ;
01B3    C1    POP    B           ;We must not have come to the end of
01B4    D1    POP    D           ;the user's stack.  Fetch the action
01B5    21    LXI H, 001FH       ;code (register C), the hours comp
01B6    1F                       ;code (register B), the minutes comp
01B7    00                       ;code (register E) and the seconds
01B8    7E    MOV    A,M         ;comp code (register D).  Load the
01B9    B8    CMP    B           ;current hours count into the accum
01BA    C2    JNZ    01A3H       ;and compare with B (hours comp code).
01BB    A3                       ;If no match, exit via 01A3H back to
01BC    01                       ;master program.  If match, fetch
01BD    2B    DCX    H           ;current minutes count to accum and
01BE    7E    MOV    A,M         ;compare to E (minutes comp code).
01BF    BB    CMP    E           ;If no match, exit to master program
01C0    C2    JNZ    01A3H       ;via 01A3H.  If match, fall through.
01C1    A3                       ;
01C2    01                       ;
01C3    2B    DCX ~~INX~~    H   ;Fetch seconds count to accum.
01C4    7E    MOV    A,M         ;
01C5    BA    CMP    D           ;Check against seconds comp code in D.
01C6    C2    JNZ    01A3H       ;If no match, exit to master program
```

```
01C7    A3                          ;via 01A3H.
01C8    01                          ;It's a MATCH!
01C9    21      LXI H, 6000H        ;Point H/L to LED address.
01CA    00                          ;
01CB    60                          ;
01CC    71      MOV   M,C           ;Send action code in register C to LEDs.
01Cd    C3      JMP   01AD          ;Exit to master program through 01ADH.
01CE    AD                          ;
01CF    01                          ;
```

In order to set up a portion of the memory as a stack, we need to initialize the stack pointer to the boundry of the new stack. The stack that we are used to has always grown downward; this is because a CALL instruction decrements the stack pointer instead of incrementing it. But loading data into memory locations in descending order is very awkward since the NXT key only increments the address. Since the stack we are setting up is only used for storing data, not return addresses, there is no reason why our stack should not grow upward. In this case we initialize the stack pointer to the bottom edge of the stack, 0040H. You are familiar with the LXI R instruction that loads a register pair with four digits of immediate data. So far we have used LXI B, LXI D, and LXI H instructions. There is one more, LXI SP, which operates in exactly the same fashion as the others except it loads the stack pointer instead of the working registers.

Once the stack pointer is initialized with the LXI SP instruction at 01A0H, it is time to set up the loop. The loop operates by checking the comparison codes against the current time. If there is no match, the program loops back to the beginning, fetches a new set of comparison codes and checks them against the current time. This process continues until the top of the stack is reached or a match is found. In any event, the program has to have a way of telling when the top of the stack has been reached. One way would be to load the memory location containing the hours comparison code with a nonreal code, say FFH. Since there is no way the clock could ever count up the hours to FFH, when the program reached this point in examining the stack, it would know that the end of the stack had been reached and it should return to the master program. The way it is done here is to monitor the contents of the stack pointer. Since there are very few instructions that manupulate the contents of the stack pointer, this is a little tricky.

We have already used DAD B in this chapter. That instruction, Double Add B, causes the four digit number in the H and L registers. The result is stored in the H/L registers. Likewise, DAD D adds the D/E registers to H/L and stores the results in H/L. DAD H adds the contents of H/L to the contents of H/L which is a tricky way of doubling the contents of those registers. There is one more of the DAD instructions, DAD SP. It adds the contents of the stack pointer to H/L and stores the result in H/L. The contents of the stack pointer are unaffected by the operation. If, as in the case of REVIEW STACK, we have set the contents of H/L to zero with an LXI H, 0000H instruction, DAD SP will have the effect of duplicating the contents of the stack pointer into the H/L register pair. Once that is done, we can MOV the contents of the L register to the accumulator and check it with a CPI A0H instruction to see if we have reached the top of the stack which occurs at 00A0H. If that occurs, the program falls out of the loop to an LXI SP, 0100H which restores the stack pointer to the value it had during the master program. A jump back to the master program provides the exit for the module.

Assuming the top of the stack has not been reached, the JNZ instruction at 01AAH takes program execution to 01B3H which is the beginning of the comparison process. POP B and POP D load the first four entries in the user's stack into the working registers of the CPU. The H/L registers are pointed to the current hours count in memory. That count is fetched to the accumulator with a MOV A,M instruction. The POP instructions caused the hours comparison code to be loaded into the B register. A CMP B instruction now checks to see if the current hours count and the hours comparison code match. If there is no match, a jump to 01A3H occurs which starts the whole process over; new entries from the user's stack are fetched, etc. If there is a match the program goes on to check the minutes and seconds counts. This is done by decrementing the

H/L pair so they point to the minutes current time memory location, fetching this data to the accumulator where it is compared against the minutes comparison code with a CMP E instruction, etc.

If all three of the comparison codes match, the program falls through to 01C9H where the H/L registers are pointed to the LEDs and the action code contained in register C sent. Finally the stack pointer is restored to 0100H and the return to the master program executed.

To check the program we have to put the links into place in the master program.

```
0112    C3    JMP    01A0H       ;
0113    A0                       ;
0114    01                       ;
```

Then we have to load some action and comparison codes into memory using the DCM mode. Lets assume that we want the LEDs to blank at 00:12:24 and at 00:12:43 we want the bottom LED to go on, and at 00:12:59 we want them all to go on and remain on. Load the following data into the memory.

```
0040    FF
0041    00
0042    12
0043    24
0044    FB
0045    00
0046    12
0047    43
0048    F1
0049    00
```

```
004A    12
004B    59
```

To shorten the wait, set the clock for 00:12. Then press CLR and EXC . You should see the LEDs turn on and off exactly as you programmed them. Try other combinations. The action codes are listed below:

LED	Action Code
Top only,	F7H
Middle only,	FDH
Bottom only,	FBH
Top and Middle	F4H
Top and Bottom	F2H
Middle and Bottom	F8H
All three	F1H

DIP SWITCH. We're really closing in now. There's only one more module to write and put into place. If you've spent much time with the clock function itself, you've probably notice by now that it is not too accurate. That's because we never took much time to tweak the loop constant in the ONE SECOND module. By changing that constant we change the number of passes through DISPLAY ONE-PASS that it takes to make up the ONE SECOND module. We can make the timing function more accurate by changing this constant, located at 0142H, using the DCM mode, but there's a far easier way. Write a program that changes it for you.

DIP SWITCH reads the settings of the DIP switches at the left-hand bottom corner of the keyboard, and changes the value of the loop constant at 0142H according to the settings of those switches. That allows you to easily speed up and slow down the clock, without interrupting the program. Load the program shown below into the computer.

DIP SWITCH

```
0150    3A    LDA      C000H      ;Load DIP switches into accum.  This
0151    00                        ;address contains other data as well
0152    C0                        ;so that will have to be masked out.
0153    E6    ANI      30H        ;Accum now contains only DIP switch
0154    30                        ;data.
0155    CA    JZ       0106H      ;If neither switch is closed, go
0156    06                        ;back to master program.
0157    01                        ;
0158    FE    CPI      30H        ;Are both switches closed?  If so, ignore
0159    30                        ;them and return to master program.
015A    CA    JZ 0106H
015B    06
015C    01
015D    FE    CPI      20H        ;Is the left switch closed?  If so
015E    20                        ;
015F    CA    JZ       0169H      ;go to speed up segment.  If not,
0160    69                        ;the only other possibility is that
0161    01                        ;the other switch is closed.
0162    21    LXI H, 0142H        ;Slow Down segment.  Point H/L to
0163    42                        ;timing constant in ONE SECOND.
0164    01                        ;
0165    34    INR      M          ;Increment timing constant, and go
0166    C3    JMP      0106H      ;back to master program.
0167    06
0168    01
0169    21    LXI H, 0142H        ;Speed Up.  Point H/L to timing constant
016A    42                        ;in ONE SECOND.
016B    01                        ;
016C    35    DCR      M          ;Decrement the timing constant and
016D    C3    JMP      0106H      ;go back to the master program.
016E    06
016F    01
```

The module operates by loading the DIP switch data into the accumulator with a LDA C000H instruction. As we shall see in the next chapter, this address also contains the data from the keyboard and the display guard circuit, so it is necessary to mask out or eliminate everything but the DIP switch data. We do this by forcing all of the bits that are not associated with the DIP switch to zero with an ANI 30H instruction. All that is left now is the DIP switch data. If neither of the switches are closed, the accumulator will now contain all zeroes. If that is the case, we need not adjust the timing constant since the user has not flipped either of the switches. The JZ 0106H takes us back to the master program if the accumulator now contains zero. This instruction operates in exactly the opposite sense from the familiar JNZ instruction. Between JZ and JNZ we can jump on either the zero or nonzero condition.

Assuming the accumulator does not contain zero, one or both of the switches must be set. If both switches are in this position, we shall assume there has been a mistake and ignore the closures. We can check that by CPI 30H which will set the zero flag only if both swithces are set. Again, we are forced to pass over the details of this instruction until we cover the binary code in Chapter 9. Basically 30H is 0011 0000 in binary. The two one's correspond to the DIP switch contacts. If both contacts are set, the accumulator will contain 0011 0000 at this point and the CPI 30H instruction will find a match and the zero flag will be set. The JZ 0106H instruction that follows the CPI will cause an exit to the master program in the same manner as before.

If the left switch has been set the accumulator will ~~not~~ now contain 0010 0000 and the CPI 20H instruction will find a match. If so, the zero flag will be set and the JZ 0169H instruction will cause a jump to the program segment written to speed up the computer. If not, the program execution falls through to the slow down program segment since this is the only remaining possibility. The slow down segment points H/L at the timing constant in ONE SECOND and then increments it with an INR M instruction.

We've already used this instruction, but in case you've forgotten it operates by incrementing any memory location H/L are pointing to. The segment is exited via a JMP 0106H back to the master program. The speed up segment is exactly the opposite except the DCR M instruction is used to decrement the timing constant. It is the complement of the INR M instruction used in the slow down segment.

Don't forget to put links to DIP SWITCH in the master program.

```
0103      C3      JMP     0150          ;
0104      50                            ;
0105      01                            ;
```

Conclusion. Besides the new instructions and techniques covered in this chapter, we spent a considerable amount of time going through the development of a program in exactly the way an experienced programmer would have done it. We hope the exercise will give you the confidence to write and try your own programs, because after all, that's what this course is all about.